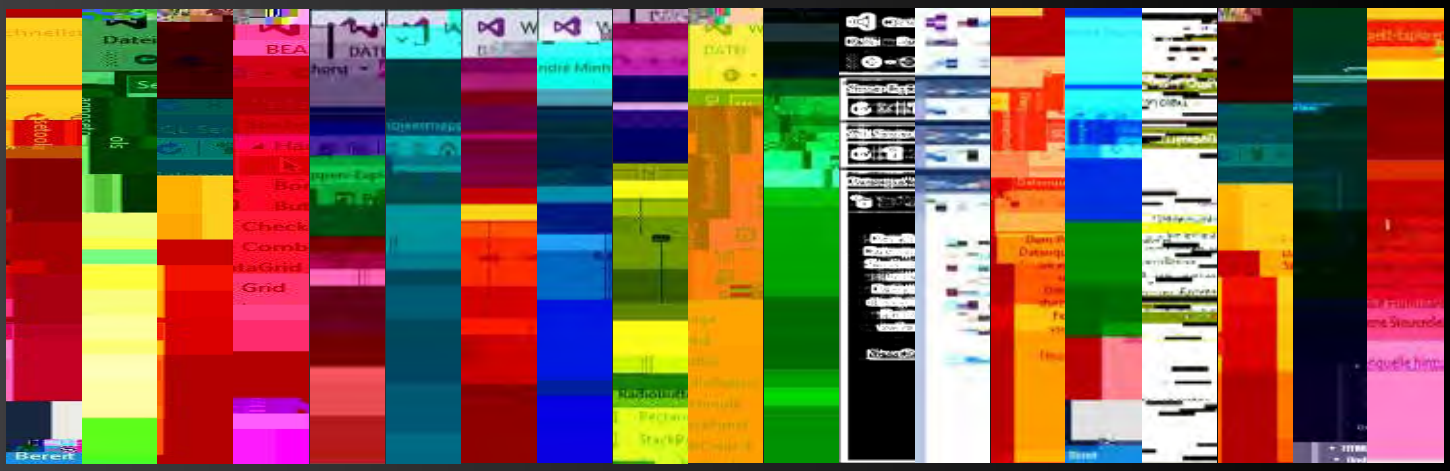
KOMPENDIUM

DATENBANK ENTWICKLER

MAGAZIN FÜR DIE DATENBANKENTWICKLUNG MIT VISUAL STUDIO FÜR DESKTOP, WEB UND CO.



KOMPENDIUM

ALLE AUSGABEN VON 2015 BIS 2022

2.400 SEITEN KNOW-HOW ZUR DATENBANKENTWICKLUNG MIT VISUAL STUDIO

ÜBER 200 BEISPIELPROJEKTE

André Minhorst Verlag

datenbankentwickler.net

André Minhorst – DATENBANKENTWICKLER Kompendium

ISBN 978-3-944216-15-7
© 2022 André Minhorst Verlag, Borkhofer Straße 17, 47137 Duisburg/Deutschland
1. Auflage 2022
Lektorat André Minhorst
Korrektur André Minhorst
Cover/Titelbild André Minhorst
Typographie, Layout und Satz André Minhorst
Herstellung André Minhorst

Das vorliegende Werk ist in all seinen Teilen urheberrechtlich geschützt. Alle Rechte vorbehalten, insbesondere das Recht der Übersetzung, des Vortrags, der Reproduktion, der Vervielfältigung auf fotomechanischem oder anderen Wegen und der Speicherung in elektronischen Medien. Ungeachtet der Sorgfalt, die auf die Erstellung von Text, Abbildungen und Programmen verwendet wurde, können weder Verlag noch Autor, Herausgeber oder Übersetzer für mögliche Fehler und deren Folgen eine juristische Verantwortung oder irgendeine Haftung übernehmen.
Die in diesem Werk wiedergegebenen Gebrauchsnamen, Handelsnamen, Warenbezeichnungen et cetera können auch ohne besondere Kennzeichnung Marken sein und als solche den gesetzlichen Bestimmungen unterliegen.
Die in den Beispielen verwendeten Namen von Firmen, Produkten, Personen oder E-Mail-Adressen sind frei erfunden, soweit nichts anderes angegeben ist. Jede Ähnlichkeit mit tatsächlichen Firmen, Produkten, Personen oder E-Mail-Adressen ist rein zufällig.

Inhalt

Einstieg in Visual Studio Community 2013 23
Von VBA zu C#: Erste Anwendung und Variablen 28
Von VBA zu C#: Operatoren 37
Von VBA zu C#: Bedingungen 42
Von VBA zu C#: Schleifen 46
Von VBA zu C#: Arrays 51
Die Console-Klasse 57
Von Access zu WPF: Fenster 61
Datenzugriff mit ADO.NET, Teil 1 71
Objektorientierte Programmierung, Teil 1 79
Mit Zeichenketten arbeiten 91
Auflistungen mit der ArrayList 102
Steuerelemente anordnen 112
Daten im DataGrid-Steuerelement anzeigen I 124
Objektorientierte Programmierung: Grundlagen II 130
Objektorientierte Programmierung: Vererbung I 141
Experimentieren mit der Konsole 147
Von VBA zu C#: Das Static-Schlüsselwort 155
Von VBA zu C#: Steuerelement-Ereignisse 158
Von VBA zu C#: Objekt-Ereignisse 163
Von VBA zu C#: Dateidialoge 170
Delegates 178
Anwendungskonfigurationsdateien 181
Datenzugriff mit ADO.NET, Teil 2 192
Fenster mit einfachen Tabellendaten 205
Methodenstarter als Vorlage 217
Bindung zwischen Steuerelementen 223
WPF-Controls: Schaltflächen 230
WPF-Controls: Kombinationsfelder 235
WPF-Controls: ToolTips 242
Fenster mit 1:n-Daten und Lookup-Feld 247
Kombinationsfelder mit Daten füllen 254
SQL Server 2014 Express installieren 260
Von der .mdb-Datei zum SQL Server 266
ADO.NET: Typisierte und untypisierte DataSets 272
Tipps und Tricks zu Fenstern und Steuerelementen 283
Von VBA zu C#: Fehlerbehandlung 291
Objektorientierte Programmierung: Interfaces 307
C#-DLL für COM/VBA erstellen 312

WPF-Datenbindung: Einfache Objekte 324
WPF-Datenbindung: Listen-Objekte 330
Einführung in das Entity Framework 339
Tipps und Tricks 353
LocalDB-Datenbanken nutzen 359
Datenbanken erstellen mit dem SSMA 364
SQL Server-Interaktion mit dem Profiler verfolgen 377
Entity Data Model für eine Datenbank erstellen 382
LINQ to Entities: Daten abfragen 391
LINQ to Entities: Daten bearbeiten 403
WPF/EDM: Kundenübersicht 407
WPF/EDM: Kundendetails 416
SQL Server: Datenbanken per Skript kopieren 423
Die MessageBox-Klasse 431
InputBox im Eigenbau 434
Ribbons mit WPF 441
EDM: Einfaches Detailfenster 454
EDM: Lookup-Kombinationsfelder 467
EDM: Kombinationsfelder erweitern 470
EDM: 1:n-Beziehung als Parent-Child-Ansicht 477
Tipps und Tricks 483
WPF-Steuerelemente: Das DockPanel 487
WPF-Trigger 490
Abhängige Eigenschaften per Binding 497
Basics: PropertyChanged 502
Events in der Praxis 506
EDM: Ausnahmen beim Speichern behandeln 515
EDM: Validieren von Entitäten mit IDataErrorInfo 521
EDM: Kunden verwalten mit Ribbon 529
EDM: Kundendetails verwalten 539
PropertyChanged in der Praxis 546
WPF-Basics: Namespaces 551
IntelliSense-Unterstützung für eigene Code-Elemente 554
Quellcodeverwaltung mit Visual Studio und Git 557
SQLite als Backend 569
Anwendung von SQL Server zu SQLite wechseln 577
Bestellverwaltung planen 580
EDM: 1:n-Beziehungen mit DataGrid 592
Tipps und Tricks zu Visual Studio 606
Datumsfunktionen 615
DataGrid im Detail 627

Webservice testen am Beispiel von DHL 634
Webservice mit C# am Beispiel von DHL-Etiketten 645
PDFs zusammenführen mit iTextSharp 655
EDM: Bestellungen und Bestellpositionen 663
Das ListBox-Steuerelement: Grundlagen 679
EDM: Datenbindung mit dem ListBox-Steuerelement 688
EDM: Bilder speichern und anzeigen 696
SQLite-Datenmodellierung per C# 706
Entity Framework: SQLite verknüpfen 720
Weitergabe von WPF/C#/SQLite-Anwendungen 729
Tipps und Tricks 740
Basics: ObservableCollection 743
Type Converter in WPF/C# 752
Fehlerhafte Bindungen prüfen 756
Neuer Eintrag in ComboBox 758
m:n-Beziehung mit Listenfeld 765
LINQ to Entities-Beispiele in der Methodensyntax 782
EDM: SQLite aktuell halten 792
EDM: Backend ändern 793
LINQPad: LINQ, C# und VB einfach ausprobieren 807
Property-Elemente am Beispiel Content 812
Drag and Drop-Grundlagen 816
Von C# zu VB: Basics, Variablen, Operatoren 825
Bubbling und Tunneling: Routed Events 835
Drag and Drop mit ListBox-Elementen 842
Excel-Export mit Spreadsheet Light 851
Bestellverwaltung á la Visual Basic 859
Drag and Drop-Grundlagen 869
Visual Studio-Vorlagen erstellen 881
Grid im Griff 893
Visual Basic: Bedingungen und Schleifen 900
E-Mails mit Outlook verschicken 911
Absender von Outlook-Mails einstellen 919
E-Mails ohne Outlook versenden 924
Mit Ressourcen arbeiten 928
Brief mit Word erstellen 934
E-Mails mit Outlook verschicken 947
Absender von Outlook-Mails einstellen 955
E-Mails ohne Outlook versenden 960
Mit Ressourcen arbeiten 964
Brief mit Word erstellen 970

Das DatePicker-Steuerelement 985
Das ListView-Steuerelement 993
ListView: Sortierungen 1000
Visual Basic: Mit Klassen programmieren 1005
EDM: Der Code First-Ansatz 1018
EDM: Code First - Datenbank erweitern 1028
Adressen verwalten 1037
Visual Studio 2017 Community Edition 1049
Interpolierte Zeichenketten 1052
Einfache ASP.NET Core-Anwendungen erstellen 1054
ASP.NET Core-Anwendung anpassen 1060
Markup mit Razor Pages 1065
Webdesign mit Bootstrap Studio 1075
E-Mails per Sendgrid verschicken 1087
Razor Pages: Von Seite zu Seite 1095
Kontaktformular unter ASP.NET Core 1106
Razor Pages mit Datenbankanbindung 1116
Authentifizierung unter ASP.NET Core 1138
ASP.NET Core: Validierung 1153
ASP.NET Core: Anwendungen veröffentlichen 1167
.NET Core auf Version 2.1 aktualisieren 1177
EDM: 1:n-Beziehungen per Code First 1179
EDM: 1:1-Beziehungen per Code First 1188
EF Core: Klassendiagramm anzeigen 1195
Schnellstart mit Datenbank 1197
Authentifizierung nachrüsten 1201
Authentifizierungsseiten anpassen 1212
Authentifizierung um Felder erweitern 1219
Razor-Pages: Daten aus Lookup-Tabellen 1229
Von Access zum Entity Framework: Basics 1241
Entity Framework: Datenbankinitialisierung 1244
Entity Framework: Datenbankmigration 1254
Von Access zu Entity Framework: Datenmodell 1264
Von Access zu Entity Framework: Daten 1281
Von Access zu EF: Step by Step 1293
EDM: DataGrid als Datenblatt 1295
SQL Server-Datenbank ins Web mit SQL Azure 1309
EDM: m:n-Beziehung per Code First 1321
SQLite Code First mit EF Core 1325
C#-DLL in VB-Projekt nutzen 1331
Einstieg in PowerApps 1334

PowerApp mit Datenbank erstellen 1345
PowerApps: Artikel verwalten 1350
PowerApps: Datensatz per DropDown auswählen 1365
PowerApps: Kamera steuern 1373
PowerApps: Bilder in Datenbank speichern 1378
PowerApps: Bilder aus der Datenbank anzeigen 1381
Von Access zu Entity Framework: Update 1 1385
Access zu WPF: Detailformulare mit Textfeldern 1395
EF: Daten abfragen mit VB und LINQ 1408
Entity Framework: Gespeicherte Prozeduren 1423
Entity Framework: Daten archivieren 1430
Bytes im Griff mit der Stream-Klasse: FileStream 1437
WPF: DataTrigger und MultiDataTrigger 1440
Mehrspaltige Kombinationsfelder 1448
Entity Framework: Bilder in WPF 1457
EDM für bestehende Datenbank mit Code First 1473
EDM: Blättern in Datensätzen 1487
Validieren mit VB und EDM 1497
Entity Framework: Der ChangeTracker 1511
Änderungen erkennen und verwerfen 1522
Detailformulare mit Combo, Checkbox und Button 1530
Access zu WPF: Validierung und Navigation 1550
FlowDocument-Elemente mit XAML 1565
Das RichTextBox-Steuerelement 1577
Format-Menü für die RichTextBox 1588
Codeschnipsel in Visual Studio 1601
Onlinebanking mit DDBAC: Testdaten 1613
Onlinebanking mit DDBAC: FinTS-Registrierungen 1614
Onlinebanking mit DDBAC II: Überweisungen 1616
Das Expander-Steuerelement 1629
Das TextBox-Steuerelement 1632
Kontextmenüs mit WPF 1640
Steuerelemente mit Stil 1650
Visual Basic: AndOr und OrElse 1657
Programmieren der Zwischenablage mit VB 1660
Access zu EDM: Dateien erstellen 1670
Access zu EDM: INotifyPropertyChanged integrieren 1678
Access zu EDM: Validierung 1684
Rechnungsverwaltung, Teil 1: Grundlagen 1693
Rechnungsverwaltung, Teil 2: Rechnungspositionen 1713
Rechnungsbericht mit XAML 1732

Rechnungsverwaltung auf SQLite umstellen 1746
Entity Framework: Aktionsabfragen mit LINQ 1757
Debugging in Visual Studio 1765
Fehlerbehandlung unter VB.NET 1778
XAML-Eigenschaften beim Debuggen testen 1788
FlowDocuments mit VB zusammenstellen 1796
XML mit .NET 1805
XML-Dokumente schnell lesen mit XmlReader 1818
Bestellungen per XmlReader einlesen 1828
XML-Dokumente erstellen mit XmlWriter 1838
Bestellungen per XmlWriter in XML exportieren 1845
XML lesen mit dem Document Object Model 1851
Onlinebanking mit DDBAC: Benutzeroberfläche 1861
Das GridSplitter-Steuerelement 1885
Das Kalendersteuerelement 1893
Commands verwenden 1900
Anwendungseinstellungen unter VB nutzen 1913
Das Menu-Steuerelement 1918
Symbolleisten mit dem ToolBar-Steuerelement 1924
Symbolleiste speichern und wiederherstellen 1933
XML schreiben mit dem Document Object Model 1937
IsDefault: Standardschaltfläche festlegen 1949
IsCancel: Abbrechen-Schaltfläche festlegen 1950
SQL Server: Fehler bei Entwurfsänderungen 1951
AdventureWorks: Schnelle Beispieldatenbank 1955
EDM mit vorhandener SQL Server-Datenbank 1957
Code First Mapping per Fluent API 1969
Code First Mapping per DataAnnotation 1981
SQL Server Reporting Services 1991
Reporting Services 2019 installieren und starten 1996
SQL Server Report Builder 2006
Reporting Services: Web-Portal 2014
Reporting Services Datenquellen 2021
Reporting Services Datasets 2030
Reporting Services Datasets mit Parametern 2045
Reporting Services: Tabellarische Berichte 2057
Reporting Services: Gruppen und Summen 2066
Das WPF-Webbrowser-Steuerelement 2077
Navigieren im Frame-Steuerelement 2085
Nicht-generische Auflistungen unter VB.NET 2094
Generische Auflistungen unter VB.NET 2104
CSV-Datei in Klassen importieren 2120

EDM-Daten in CSV exportieren 2131
Von Page zu Page 2141
Fensterposition beim Öffnen einstellen 2151
Daten in Listen filtern mit ListCollectionView 2157
Das WPF-Steuerelement CheckBox 2171
Optionsgruppen mit GroupBox und RadioButtons 2181
Optionsgruppe an Boolean binden 2188
Markierung bei Fokuserhalt der TextBox 2190
AfterUpdate-Ereignis für TextBox-Elemente 2195
Typumwandlung unter VB.NET 2200
Navigieren mit CollectionViewSource 2209
Versionsverwaltung mit Git 2219
Beispieldaten generieren mit Bogus 2231
Beispieldaten für ein EDM generieren 2242
Seminarverwaltung I: Entity Data Model 2253
Seminarverwaltung II: Ribbon und Frame 2259
Seminarverwaltung III: Daten vom Shopsystem 2262
TabControl: Das Registersteuerelement von WPF 2273
Registerseiten im TabControl an Daten binden 2283
Unit-Testing mit Visual Studio 2291
Seminarverwaltung IVa: Kunden und Seminare 2303
Seminarverwaltung IVb: Kunden und Seminare 2318
Bitmaps programmieren mit VB.NET 2337
Zeichnen in Bitmaps mit der Graphics-Klasse 2349
Texte in Bitmaps einfügen 2366
EPC-QR-Code für Rechnungen 2374
EPC-QR-Code per DLL 2385
Setup für COM-DLLs mit Inno Setup 2396

Inhalt nach Kategorien

Anwendungsentwicklung

Anwendungskonfigurationsdateien 181
Weitergabe von WPF/C#/SQLite-Anwendungen 729
Mit Ressourcen arbeiten 964
Von Access zum Entity Framework: Basics 1241
Anwendungseinstellungen unter VB nutzen 1913
Setup für COM-DLLs mit Inno Setup 2396

ASP.NET Core

Einfache ASP.NET Core-Anwendungen erstellen 1054
ASP.NET Core-Anwendung anpassen 1060
Markup mit Razor Pages 1065
Webdesign mit Bootstrap Studio 1075
E-Mails per Sendgrid verschicken 1087
Razor Pages: Von Seite zu Seite 1095
Kontaktformular unter ASP.NET Core 1106
Razor Pages mit Datenbankanbindung 1116
Authentifizierung unter ASP.NET Core 1138
ASP.NET Core: Validierung 1153
ASP.NET Core: Anwendungen veröffentlichen 1167
.NET Core auf Version 2.1 aktualisieren 1177
Schnellstart mit Datenbank 1197
Authentifizierung nachrüsten 1201
Authentifizierungsseiten anpassen 1212
Authentifizierung um Felder erweitern 1219
Razor-Pages: Daten aus Lookup-Tabellen 1229

Benutzeroberfläche mit WPF

Von Access zu WPF: Fenster 61
Steuerelemente anordnen 112
Daten im DataGrid-Steuerelement anzeigen I 124
Fenster mit einfachen Tabellendaten 205
WPF-Controls: Schaltflächen 230
WPF-Controls: Kombinationsfelder 235
WPF-Controls: ToolTips 242

Fenster mit 1:n-Daten und Lookup-Feld 247
Kombinationsfelder mit Daten füllen 254
Tipps und Tricks zu Fenstern und Steuerelementen 283
WPF/EDM: Kundenübersicht 407
WPF/EDM: Kundendetails 416
InputBox im Eigenbau 434
Ribbons mit WPF 441
EDM: Einfaches Detailfenster 454
EDM: Lookup-Kombinationsfelder 467
EDM: Kombinationsfelder erweitern 470
EDM: 1:n-Beziehung als Parent-Child-Ansicht 477
WPF-Steuerelemente: Das DockPanel 487
EDM: Kunden verwalten mit Ribbon 529
EDM: Kundendetails verwalten 539
PropertyChanged in der Praxis 546
Bestellverwaltung planen 580
EDM: 1:n-Beziehungen mit DataGrid 592
DataGrid im Detail 627
Das ListBox-Steuerelement: Grundlagen 679
Neuer Eintrag in ComboBox 758
m:n-Beziehung mit Listenfeld 765
Drag and Drop-Grundlagen 816
Drag and Drop mit ListBox-Elementen 842
Bestellverwaltung á la Visual Basic 859
Drag and Drop-Grundlagen 869
Grid im Griff 893
EDM: DataGrid als Datenblatt 1295
Mehrspaltige Kombinationsfelder 1448
Entity Framework: Bilder in WPF 1457
EDM: Blättern in Datensätzen 1487
Änderungen erkennen und verwerfen 1522
Access zu WPF: Validierung und Navigation 1550
FlowDocument-Elemente mit XAML 1565
FlowDocuments mit VB zusammenstellen 1796
Das Menu-Steuerelement 1918
Symbolleisten mit dem ToolBar-Steuerelement 1924
Symbolleiste speichern und wiederherstellen 1933
Navigieren im Frame-Steuerelement 2085
Von Page zu Page 2141
Fensterposition beim Öffnen einstellen 2151
Daten in Listen filtern mit ListCollectionView 2157

Navigieren mit CollectionViewSource 2209

Berichte und Reporting

Rechnungsbericht mit XAML 1732
SQL Server Reporting Services 1991
Reporting Services 2019 installieren und starten 1996
SQL Server Report Builder 2006
Reporting Services: Web-Portal 2014
Reporting Services Datenquellen 2021
Reporting Services Datasets 2030
Reporting Services Datasets mit Parametern 2045
Reporting Services: Tabellarische Berichte 2057
Reporting Services: Gruppen und Summen 2066

C#-Grundlagen

Von VBA zu C#: Erste Anwendung und Variablen 28
Von VBA zu C#: Operatoren 37
Von VBA zu C#: Bedingungen 42
Von VBA zu C#: Schleifen 46
Von VBA zu C#: Arrays 51
Mit Zeichenketten arbeiten 91
Auflistungen mit der ArrayList 102
Von VBA zu C#: Das Static-Schlüsselwort 155
Von VBA zu C#: Steuerelement-Ereignisse 158
Von VBA zu C#: Objekt-Ereignisse 163
Von VBA zu C#: Dateidialoge 170
Von VBA zu C#: Fehlerbehandlung 291
Die MessageBox-Klasse 431
WPF-Trigger 490
Basics: PropertyChanged 502
Events in der Praxis 506
IntelliSense-Unterstützung für eigene Code-Elemente 554
Datumsfunktionen 615
Basics: ObservableCollection 743
Type Converter in WPF/C# 752
Bubbling und Tunneling: Routed Events 835
Interpolierte Zeichenketten 1052
XML lesen mit dem Document Object Model 1851

C#-Klassen und Bibliotheken

Die Console-Klasse 57
E-Mails ohne Outlook versenden 960
C#-DLL in VB-Projekt nutzen 1331

C#-Programmiertechnik

Objektorientierte Programmierung, Teil 1 79
Objektorientierte Programmierung: Grundlagen II 130
Objektorientierte Programmierung: Vererbung I 141
Delegates 178
Objektorientierte Programmierung: Interfaces 307
C#-DLL für COM/VBA erstellen 312
PDFs zusammenführen mit iTextSharp 655

Datenzugriffstechnik

Datenzugriff mit ADO.NET, Teil 1 71
Datenzugriff mit ADO.NET, Teil 2 192
SQL Server 2014 Express installieren 260
Von der .mdb-Datei zum SQL Server 266
ADO.NET: Typisierte und untypisierte DataSets 272
WPF-Datenbindung: Einfache Objekte 324
WPF-Datenbindung: Listen-Objekte 330
LocalDB-Datenbanken nutzen 359
EDM: Ausnahmen beim Speichern behandeln 515
EDM: Validieren von Entitäten mit IDataErrorInfo 521
EDM: Bestellungen und Bestellpositionen 663
SQLite-Datenmodellierung per C# 706
LINQ to Entities-Beispiele in der Methodensyntax 782
EDM: Der Code First-Ansatz 1018
EF: Daten abfragen mit VB und LINQ 1408
Beispieldaten generieren mit Bogus 2231
Beispieldaten für ein EDM generieren 2242

Entity Framework

Einführung in das Entity Framework 339
Entity Data Model für eine Datenbank erstellen 382
LINQ to Entities: Daten abfragen 391
LINQ to Entities: Daten bearbeiten 403

EDM: Einfaches Detailfenster 454
EDM: Lookup-Kombinationsfelder 467
EDM: Kombinationsfelder erweitern 470
EDM: 1:n-Beziehung als Parent-Child-Ansicht 477
EDM: Ausnahmen beim Speichern behandeln 515
EDM: Validieren von Entitäten mit IDataErrorInfo 521
EDM: Datenbindung mit dem ListBox-Steuerelement 688
EDM: Bilder speichern und anzeigen 696
Entity Framework: SQLite verknüpfen 720
LINQ to Entities-Beispiele in der Methodensyntax 782
EDM: SQLite aktuell halten 792
EDM: Backend ändern 793
EDM: Der Code First-Ansatz 1018
EDM: Code First - Datenbank erweitern 1028
Adressen verwalten 1037
EDM: 1:n-Beziehungen per Code First 1179
EDM: 1:1-Beziehungen per Code First 1188
EF Core: Klassendiagramm anzeigen 1195
Entity Framework: Datenbankinitialisierung 1244
Entity Framework: Datenbankmigration 1254
Von Access zu Entity Framework: Datenmodell 1264
Von Access zu Entity Framework: Daten 1281
Von Access zu EF: Step by Step 1293
EDM: m:n-Beziehung per Code First 1321
Von Access zu Entity Framework: Update 1 1385
EF: Daten abfragen mit VB und LINQ 1408
Entity Framework: Gespeicherte Prozeduren 1423
Entity Framework: Daten archivieren 1430
EDM für bestehende Datenbank mit Code First 1473
EDM: Blättern in Datensätzen 1487
Validieren mit VB und EDM 1497
Entity Framework: Der ChangeTracker 1511
Entity Framework: Aktionsabfragen mit LINQ 1757
EDM mit vorhandener SQL Server-Datenbank 1957
Code First Mapping per Fluent API 1969
Code First Mapping per DataAnnotation 1981

Entity Framework Core

SQLite Code First mit EF Core 1325

Interaktiv

C#-DLL für COM/VBA erstellen 312
Webservice testen am Beispiel von DHL 634
Webservice mit C# am Beispiel von DHL-Etiketten 645
Excel-Export mit Spreadsheet Light 851
E-Mails mit Outlook verschicken 911
Absender von Outlook-Mails einstellen 919
E-Mails ohne Outlook versenden 924
Brief mit Word erstellen 934
E-Mails mit Outlook verschicken 947
Absender von Outlook-Mails einstellen 955
Brief mit Word erstellen 970
E-Mails ohne Outlook versenden 960
XML mit .NET 1805
XML-Dokumente schnell lesen mit XmlReader 1818
Bestellungen per XmlReader einlesen 1828
XML-Dokumente erstellen mit XmlWriter 1838
Bestellungen per XmlWriter in XML exportieren 1845
XML schreiben mit dem Document Object Model 1937
CSV-Datei in Klassen importieren 2120
EDM-Daten in CSV exportieren 2131

Lösungen

EDM: Kunden verwalten mit Ribbon 529
EDM: Kundendetails verwalten 539
PropertyChanged in der Praxis 546
Bestellverwaltung planen 580
EDM: 1:n-Beziehungen mit DataGrid 592
PDFs zusammenführen mit iTextSharp 655
Bestellverwaltung á la Visual Basic 859
EDM: Code First - Datenbank erweitern 1028
Adressen verwalten 1037
Onlinebanking mit DDBAC: Testdaten 1613
Onlinebanking mit DDBAC: FinTS-Registrierungen 1614
Onlinebanking mit DDBAC II: Überweisungen 1616
Rechnungsverwaltung, Teil 1: Grundlagen 1693
Rechnungsverwaltung, Teil 2: Rechnungspositionen 1713
Onlinebanking mit DDBAC: Benutzeroberfläche 1861
Seminarverwaltung I: Entity Data Model 2253
Seminarverwaltung II: Ribbon und Frame 2259

Seminarverwaltung IVa: Kunden und Seminare 2303
Seminarverwaltung IVb: Kunden und Seminare 2318
EPC-QR-Code für Rechnungen 2374
EPC-QR-Code per DLL 2385
Seminarverwaltung III: Daten vom Shopsystem 2262

News und Tools

LINQPad: LINQ, C# und VB einfach ausprobieren 807
.NET Core auf Version 2.1 aktualisieren 1177

PowerApps

Einstieg in PowerApps 1334
PowerApp mit Datenbank erstellen 1345
PowerApps: Artikel verwalten 1350
PowerApps: Datensatz per DropDown auswählen 1365
PowerApps: Kamera steuern 1373
PowerApps: Bilder in Datenbank speichern 1378
PowerApps: Bilder aus der Datenbank anzeigen 1381

SQL Server und Co.

SQL Server 2014 Express installieren 260
Von der .mdb-Datei zum SQL Server 266
LocalDB-Datenbanken nutzen 359
Datenbanken erstellen mit dem SSMA 364
SQL Server-Interaktion mit dem Profiler verfolgen 377
SQL Server: Datenbanken per Skript kopieren 423
SQLite als Backend 569
Anwendung von SQL Server zu SQLite wechseln 577
SQLite-Datenmodellierung per C# 706
Entity Framework: SQLite verknüpfen 720
EDM: SQLite aktuell halten 792
EDM: Backend ändern 793
SQL Server-Datenbank ins Web mit SQL Azure 1309
SQLite Code First mit EF Core 1325
Entity Framework: Gespeicherte Prozeduren 1423
Rechnungsverwaltung auf SQLite umstellen 1746
SQL Server: Fehler bei Entwurfsänderungen 1951
AdventureWorks: Schnelle Beispieldatenbank 1955

Tipps und Tricks

Experimentieren mit der Konsole 147
Methodenstarter als Vorlage 217
Tipps und Tricks zu Fenstern und Steuerelementen 283
Tipps und Tricks 353
Tipps und Tricks 483
Tipps und Tricks 740
Tipps und Tricks zu Visual Studio 606

VB-Grundlagen

Von C# zu VB: Basics, Variablen, Operatoren 825
Visual Basic: Bedingungen und Schleifen 900
Visual Basic: Mit Klassen programmieren 1005
C#-DLL in VB-Projekt nutzen 1331
Bytes im Griff mit der Stream-Klasse: FileStream 1437
Validieren mit VB und EDM 1497
Visual Basic: AndOr und OrElse 1657
Programmieren der Zwischenablage mit VB 1660
Fehlerbehandlung unter VB.NET 1778
Nicht-generische Auflistungen unter VB.NET 2094
Generische Auflistungen unter VB.NET 2104
Typumwandlung unter VB.NET 2200
Unit-Testing mit Visual Studio 2291
Bitmaps programmieren mit VB.NET 2337
Zeichnen in Bitmaps mit der Graphics-Klasse 2349
Texte in Bitmaps einfügen 2366

Visual Studio nutzen

Einstieg in Visual Studio Community 2013 23
Quellcodeverwaltung mit Visual Studio und Git 557
Tipps und Tricks zu Visual Studio 606
Visual Studio-Vorlagen erstellen 881
Visual Studio 2017 Community Edition 1049
Codeschnipsel in Visual Studio 1601
Debugging in Visual Studio 1765
Versionsverwaltung mit Git 2219

Von Access zu .NET

Von VBA zu C#: Erste Anwendung und Variablen 28
Von VBA zu C#: Operatoren 37
Von VBA zu C#: Bedingungen 42
Von VBA zu C#: Schleifen 46
Von VBA zu C#: Arrays 51
Von der .mdb-Datei zum SQL Server 266
Von Access zum Entity Framework: Basics 1241
Von Access zu Entity Framework: Datenmodell 1264
Von Access zu Entity Framework: Daten 1281
Von Access zu EF: Step by Step 1293
EDM: DataGrid als Datenblatt 1295
Von Access zu Entity Framework: Update 1 1385
EDM für bestehende Datenbank mit Code First 1473
Detailformulare mit Combo, Checkbox und Button 1530
Access zu EDM: Dateien erstellen 1670
Access zu EDM: INotifyPropertyChanged integrieren 1678
Access zu EDM: Validierung 1684
Access zu WPF: Validierung und Navigation 1550

Von Access zu WPF

Von Access zu WPF: Fenster 61
Daten im DataGrid-Steuerelement anzeigen I 124
Fenster mit einfachen Tabellendaten 205
Fenster mit 1:n-Daten und Lookup-Feld 247
Kombinationsfelder mit Daten füllen 254
WPF/EDM: Kundenübersicht 407
WPF/EDM: Kundendetails 416
Ribbons mit WPF 441
EDM: Kunden verwalten mit Ribbon 529
EDM: Kundendetails verwalten 539
Access zu WPF: Detailformulare mit Textfeldern 1395

Web-Apps: Razor Pages

Einfache ASP.NET Core-Anwendungen erstellen 1054
ASP.NET Core-Anwendung anpassen 1060
Markup mit Razor Pages 1065
Webdesign mit Bootstrap Studio 1075
E-Mails per Sendgrid verschicken 1087

Razor Pages: Von Seite zu Seite 1095
Kontaktformular unter ASP.NET Core 1106
Razor Pages mit Datenbankanbindung 1116
Authentifizierung unter ASP.NET Core 1138
ASP.NET Core: Validierung 1153
ASP.NET Core: Anwendungen veröffentlichen 1167
Razor-Pages: Daten aus Lookup-Tabellen 1229

WPF-Grundlagen

Bindung zwischen Steuerelementen 223
WPF-Datenbindung: Einfache Objekte 324
WPF-Datenbindung: Listen-Objekte 330
WPF-Steuerelemente: Das DockPanel 487
WPF-Trigger 490
Abhängige Eigenschaften per Binding 497
WPF-Basics: Namespaces 551
EDM: Datenbindung mit dem ListBox-Steuerelement 688
EDM: Bilder speichern und anzeigen 696
Fehlerhafte Bindungen prüfen 756
Property-Elemente am Beispiel Content 812
WPF: DataTrigger und MultiDataTrigger 1440
XAML-Eigenschaften beim Debuggen testen 1788
Commands verwenden 1900
IsDefault: Standardschaltfläche festlegen 1949
IsCancel: Abbrechen-Schaltfläche festlegen 1950
Type Converter in WPF/C# 752

WPF-Steuerelemente

Von VBA zu C#: Steuerelement-Ereignisse 158
WPF-Controls: Schaltflächen 230
WPF-Controls: Kombinationsfelder 235
WPF-Controls: ToolTips 242
EDM: Lookup-Kombinationsfelder 467
EDM: Kombinationsfelder erweitern 470
WPF-Steuerelemente: Das DockPanel 487
EDM: 1:n-Beziehungen mit DataGrid 592
DataGrid im Detail 627
Das ListBox-Steuerelement: Grundlagen 679
EDM: Datenbindung mit dem ListBox-Steuerelement 688

EDM: Bilder speichern und anzeigen 696
Neuer Eintrag in ComboBox 758
m:n-Beziehung mit Listenfeld 765
Drag and Drop mit ListBox-Elementen 842
Grid im Griff 893
Das DatePicker-Steuerelement 985
Das ListView-Steuerelement 993
ListView: Sortierungen 1000
EDM: DataGrid als Datenblatt 1295
Mehrspaltige Kombinationsfelder 1448
Entity Framework: Bilder in WPF 1457
Detailformulare mit Combo, Checkbox und Button 1530
Das RichTextBox-Steuerelement 1577
Format-Menü für die RichTextBox 1588
Das Expander-Steuerelement 1629
Das TextBox-Steuerelement 1632
Kontextmenüs mit WPF 1640
Steuerelemente mit Stil 1650
Das Kalendersteuerelement 1893
Das GridSplitter-Steuerelement 1885
Das WPF-Webbrowser-Steuerelement 2077
Das WPF-Steuerelement CheckBox 2171
Optionsgruppen mit GroupBox und RadioButtons 2181
Optionsgruppe an Boolean binden 2188
Markierung bei Fokuserhalt der TextBox 2190
AfterUpdate-Ereignis für TextBox-Elemente 2195
TabControl: Das Registersteuerelement von WPF 2273
Registerseiten im TabControl an Daten binden 2283

AUSGABE 01/2015

DATENBANK ENTWICKLER

MAGAZIN FÜR DIE DATENBANKENTWICKLUNG MIT VISUAL STUDIO FÜR DESKTOP, WEB UND CO.

TOP-THEMEN:

VISUAL STUDIO	Einstieg in Visual Studio Community 2013	SEITE 3
C#-GRUNDLAGEN	Von VBA zu C#	SEITE 8
WPF	Von Access zu WPF: Fenster	SEITE 41
DATENZUGRIFF	Datenzugriff mit ADO.NET, Teil 1	SEITE 51
PROGRAMMIEREN	Objektorientierte Programmierung, Teil 1	SEITE 59

André Minhorst Verlag

Einstieg in Visual Studio Community 2013

Mit Visual Studio Community 2013 bietet Microsoft für Selbstständige und kleine Unternehmen eine kostenlose Version seiner Entwicklungsumgebung an, die im Gegensatz zu den Express Editionen vorheriger Versionen keine Einschränkungen aufweist. In der Artikelkategorie »Visual Studio nutzen« stellen wir die Entwicklungsumgebung vor und zeigen nicht nur, wie Sie den Übergang vom VBA-Editor schaffen. Im ersten Teil beginnen wir mit der Installation und ersten Schritten.

Visual Studio Community 2013 installieren

Die Installation birgt keine größeren Herausforderungen in sich. Sie benötigen ein Microsoft-Konto und können die Installation dann direkt von der entsprechenden Microsoft-Webseite ausführen oder die Installationsdateien als DVD-Image herunterladen. Zur Installationsseite finden Sie, da sich die URLs von Zeit zu Zeit ändern, am einfachsten über den Suchbegriff **Microsoft Visual Studio Community 2013 installieren**.

Sprache einstellen

Wenn Sie nicht die englische, sondern die deutsche Version verwenden möchten, sollten Sie gleich noch das entsprechende Language Pack herunterladen und installieren. Wenn Sie danach Visual Studio starten, wählen Sie den Menüeintrag **Extras|Optionen** aus und wechseln im nun erscheinenden Dialog **Optionen** zum Bereich **Internationale Einstellungen**. Hier finden Sie alle aktuell installierten Sprachen vor (siehe Bild 1).

Alternativen zur Entwicklungsumgebung

Theoretisch könnten Sie auch in einem Texteditor entwickeln. Die notwendigen Werkzeuge zum Kompilieren des Codes liefert das Framework mit, Sie müssten diese dann per Eingabeaufforderung steuern. Das macht natürlich keinen Spaß, also beschreiben wir an dieser Stelle auch gar nicht, wie das funktioniert. Zu den Alternativen für andere Betriebssysteme wie Linux, Unix oder OS X kommen wir in späteren Ausgaben.

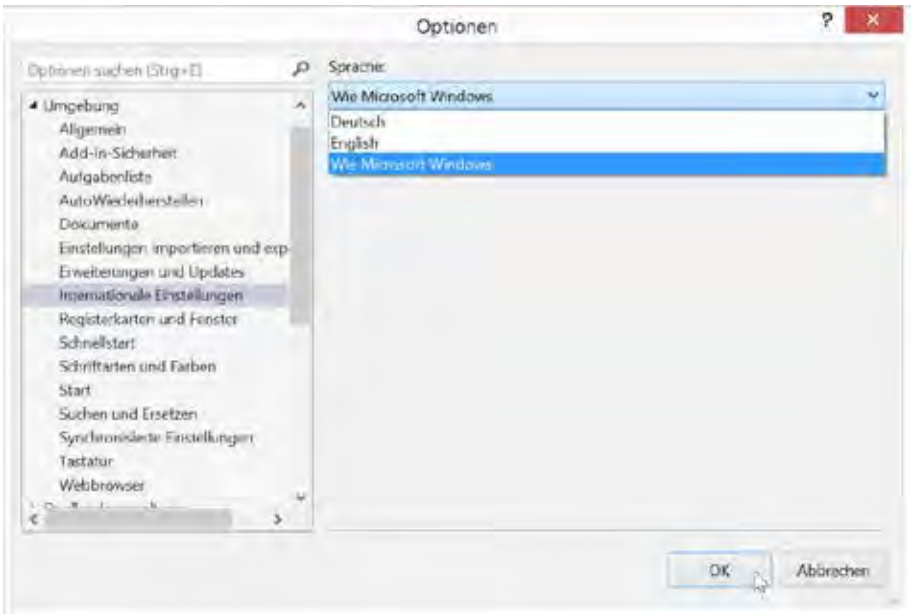


Bild 1: Einstellen der Sprache der Entwicklungsumgebung

Startbildschirm

Nach dem Öffnen von Visual Studio erscheint der Startbildschirm, der etwa die Möglichkeit zum Erstellen oder Öffnen von Projekten anbietet. Außerdem finden Sie hier Informationen über die Entwicklungsumgebung oder Neuigkeiten rund um Visual Studio. Wir wollen uns nicht damit aufhalten, sondern für unsere Aktionen die entsprechenden Menübefehle nutzen.

Elemente der Entwicklungsumgebung

Je nachdem, was für ein Projekt Sie erstellen, zeigt die

Entwicklungsumgebung verschiedene Fensterbereiche an. Diese unterscheiden sich zumeist nur unwesentlich – so erscheint etwa nach dem Erstellen einer Konsolen-Anwendung das Code-Fenster der Hauptklasse, nach dem Erstellen einer WPF-Anwendung finden Sie statt des Code-Editors den XAML-Designer zum Definieren des ersten Formulars vor. Uns interessieren die folgenden Elemente:

- Code-Editor
- Editor für die Benutzeroberfläche von Formularen
- Projektmappen-Explorer
- Eigenschaftsfenster
- Werkzeugkasten
- Fehlerliste

Einige dieser Elemente werden gleich zu Beginn komplett eingeblendet, andere sind über die Seitenleisten erreichbar (siehe Bild 2).

Wenn Sie etwa den Werkzeugkasten einblenden möchten, um die Steuerelemente für die WPF-Formulare ständig greifbar zu haben, klicken Sie links auf den vertikal angeordneten Registerreiter **Werkzeugkasten** und dann nach dem Einblenden des Werkzeugkastens auf das Pin-Icon oben rechts. Das Ergebnis sieht dann etwa wie in Bild 3 aus – der Werkzeugkasten ist nun ständig verfügbar.

Der Projektmappen-Explorer

Der Projektmappen-Explorer sieht etwa wie der Projekt-Explorer im VBA-Editor aus. Allerdings sind in Visual Studio

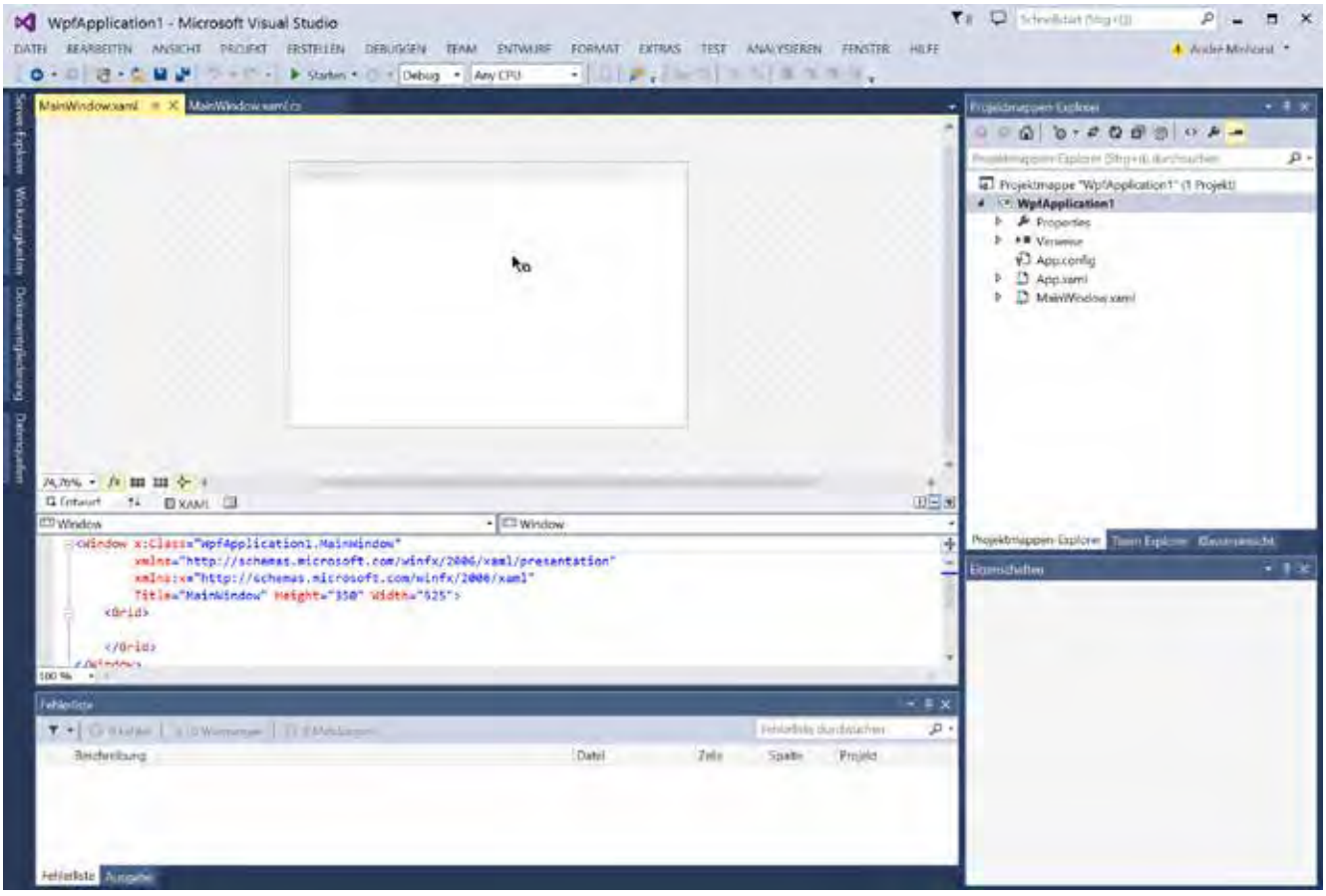


Bild 2: Standardansicht der Entwicklungsumgebung beim Bearbeiten eines WPF-Formulars

Von VBA zu C#: Erste Anwendung und Variablen

Wer von VBA zu C# wechselt, muss an einigen Stellen umdenken. Der Sprung von VBA zu Visual Basic unter .NET wäre möglicherweise etwas einfacher, allerdings ist C# wohl die weiter verbreitete Sprache. Das bedeutet nicht nur, dass es mehr Entwickler gibt, die damit entwickelte Anwendungen leichter weiterentwickeln oder Sie unterstützen können, sondern es existieren auch mehr Beispiele zu C# im Internet als zu Visual Basic. Im ersten Artikel der Kategorie »Von VBA zu C#« schauen wir uns an, wie Sie eine erste kleine Konsolenanwendung bauen und welche einfachen Datentypen es dort gibt.

Für die Beispiele dieser Artikelreihe wollen wir jeweils eine Konsolenanwendung verwenden. Damit können wir uns auf die Sprache selbst konzentrieren und müssen uns nicht mit zusätzlichen Elementen wie etwa der Benutzeroberfläche beschäftigen. Eine Konsolenanwendung erstellen Sie über den Menübefehl **Datei|Neu|Projekt**. Dies zeigt den Dialog **Neues Projekt** an, wo Sie das Zielverzeichnis und den Namen des Projekts angeben sowie die Projektvorlage auswählen – in diesem Fall **Vorlagen|Visual C#|Windows-Desktop|Konsolenanwendung**.

Ein Klick auf die Schaltfläche **OK** legt dann ein neues Projekt an, das im Wesentlichen aus einem Klassenmodul namens **Program.cs** besteht. Klassenmodule sind Ihnen, wenn Sie bereits mit VBA programmiert haben, ein Begriff – dort gab es die zu Formularen oder Berichten gehörenden Klassenmodule oder alleinstehende Klassenmodule, die Sie ähnlich wie Standardmodule angelegt haben.

Unter VBA war der wesentliche Unterschied zwischen Klassenmodulen und Standardmodulen, dass Sie zunächst ein Objekt auf Basis des Klassenmoduls erstellen mussten, um auf die enthaltenen Elemente wie etwa Methoden oder Eigenschaften zuzugreifen, während Sie die in Standardmodulen enthaltenen Prozeduren oder Funktionen direkt etwa über den Direktbereich des VBA-Editors aufrufen konnten.

Von der Idee der Standardmodule können Sie sich nun verabschieden, denn unter C# gibt es nur noch Klassenmodule. In unserem neuen Projekt heißt das aktuell einzige Exemplar **Program.cs** und sieht wie in Bild 1 aus. Was es mit den **using**-Anweisungen und den Elementen **namespace**, **class** und **static void** auf sich hat, schauen wir uns weiter unten an.

Erster Test

Die Konsolenanwendung heißt so, weil die Ausgabe in der Konsole erfolgt, also in einem Fenster wie der Eingabeaufforderung. Um dort einen Text auszugeben und das Fenster durch Betätigen der Eingabetaste wieder zu schließen,

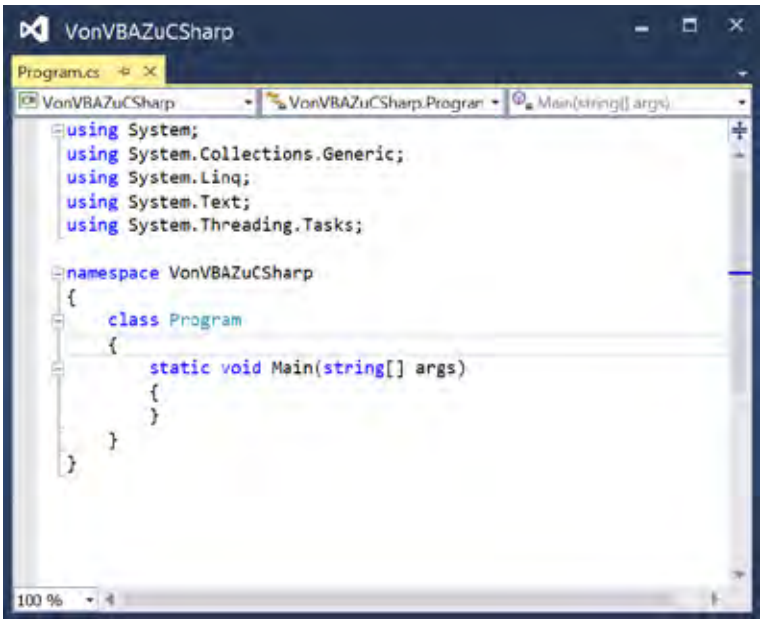


Bild 1: Standardmäßig vorhandenes Klassenmodul in einem neuen C#-Projekt

ergänzen Sie die Prozedur, die mit der Zeile **static void Main...** beginnt, durch zwei Zeilen, die Sie wie folgt zwischen den geschweiften Klammern einfügen:

```
static void Main(string[] args)
{
    Console.WriteLine("Eingabetaste drücken!");
    Console.ReadLine();
}
```

Ein Klick auf die Taste **F5** oder die Auswahl des Menüeintrags **Debuggen|Debugging starten** sorgt für die Anzeige des Fensters aus Bild 2, das Sie durch Betätigen der Eingabetaste wieder schließen.

Der erste Befehl, die Methode **WriteLine** des Objekts **Console**, hat also zunächst für die Anzeige des Textes in der Konsole gesorgt. Damit die Konsole danach nicht direkt wieder verschwindet, haben wir die **ReadLine**-Methode des **Console**-Objekts hinterhergeschickt. Diese unterbricht den Ablauf des Programms, bis der Benutzer die Eingabetaste betätigt.

Alternativ zu **ReadLine** hätten wir auch die Methode **ReadKey** nutzen können, die auf die Betätigung einer jeden Taste reagiert:

```
Console.ReadKey();
```

Semikolon als Zeilenabschluss

Sie haben es schon erkannt: Unter C# müssen Sie jede Anweisung mit einem Semikolon beenden. Es reicht nicht aus, wie unter VBA einfach die neue Anweisung in die nächste Zeile zu schreiben. Hier ist das Semikolon das Maß aller Dinge bei der Kennzeichnung des Endes einer Anweisung. Dafür können Sie unter C# auch mehrere Anweisungen in eine Zeile schreiben – Sie müssen diese nur durch das Semikolon voneinander trennen.

Dies sähe dann für unser Beispiel so aus:

Bild 2: Anzeige der Konsole mit dem gewünschten Text

```
Console.WriteLine("Eingabe...!");Console.ReadLine();
```

Zeilen umbrechen

Durch den zwingenden Zeilenabschluss per Semikolon können Sie Zeile im Gegensatz zu VBA etwas komfortabler aufteilen – zum Beispiel so:

```
Console.WriteLine(
    "Eingabetaste drücken!");
```

Sie müssen also nicht etwa das Unterstrich-Zeichen angeben, um den Zeilenumbruch zu markieren. Leider gelingt das nicht mitten in einem Literal, also innerhalb von Anführungszeichen. Folgendes führt zu einem Kompilierfehler, der übrigens rot unterstrichen markiert wird:

```
Console.WriteLine("Eingabetaste
    drücken!");
```

Außerdem können Sie sich bei dieser Gelegenheit gleich mit der Fehlerliste von Visual Studio bekanntmachen, die Sie sicher noch häufiger konsultieren werden – und zwar mit dem Menüeintrag **Ansicht|Fehlerliste** oder der Tastenkombination **Strg + ^, E** (also erst **Strg + ^**, dann **E**).

Ungewohnte Klammern

Nicht nur, dass Sie sich mit geschweiften Klammern herumschlagen müssen, die einen Codeblock beispielsweise innerhalb eines Namespaces, einer Methode oder auch eines Konstrukts wie einer Schleife oder einer Bedingung markieren – C# verlangt auch nach allen möglichen Methoden nach der Angabe eines Klammernpaares. Dies geschieht unabhängig davon, ob Parameter zu übergeben sind oder nicht. Sollten Ihnen also einmal ein Syntaxfehler

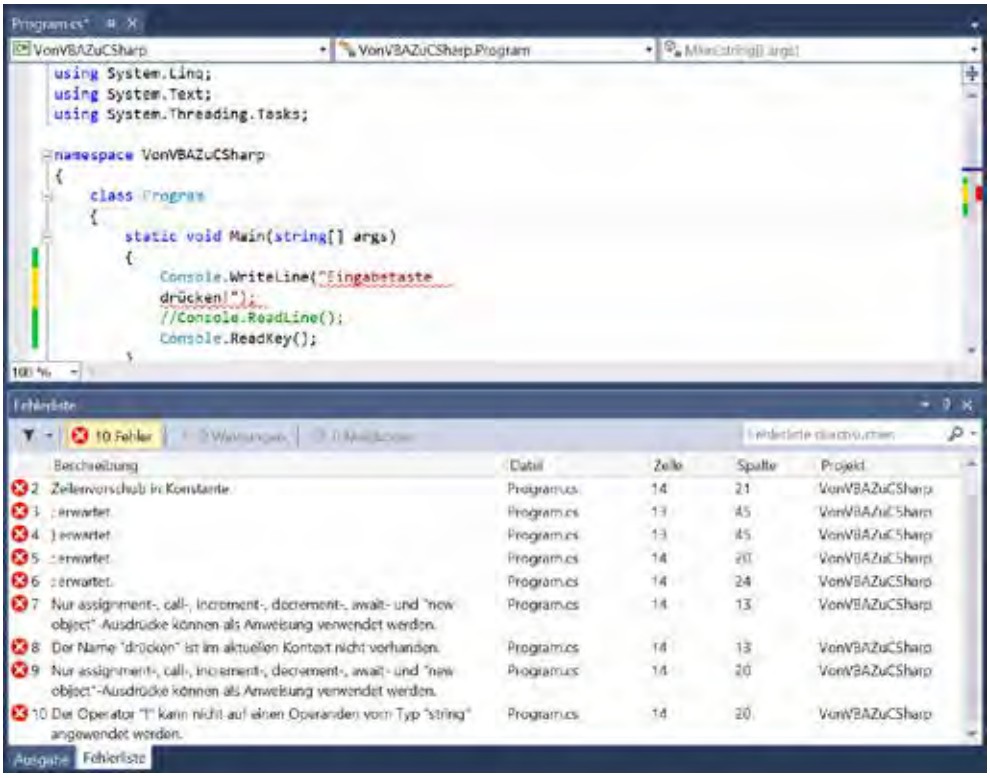


Bild 3: Fehlermeldungen

Fehler es sich handelt. Im Gegensatz zum VBA-Editor, wo etwa beim Debuggen nur ein Syntaxfehler nach dem anderen moniert wird, finden Sie hier gleich eine Liste aller aktuell vorliegenden Syntaxfehler.

Noch schöner ist, dass Sie per Doppelklick auf einen der Einträge direkt zu der fraglichen Stelle im Code springen können. Darüber hinaus zeigt die Fehlerliste auch die Datei (also in diesem Fall die Klasse), die Zeile und die Spalte des Fehlers an. Sie könnten sich also auch über die Zeilennummer den Fehler anschauen – vorausgesetzt, dass das Codefenster gerade über den Weg laufen, den Sie sich auf den ersten Blick nicht erklären können, prüfen Sie, ob die Anweisung eventuell nach einem Paar abschließender Klammern verlangt.

Fehlerliste

Diese sieht in diesem Fall wie in Bild 3 aus. Mehr Syntaxfehler kann man wohl kaum per Einzeiler provozieren. Sie können diese Anzeige jedoch zum Anlass für die erste Erkundung der Fehlerliste nutzen, die es im VBA-Editor ja gar nicht gibt: Diese zeigt genau, wo der Fehler vorliegt und um was für einen

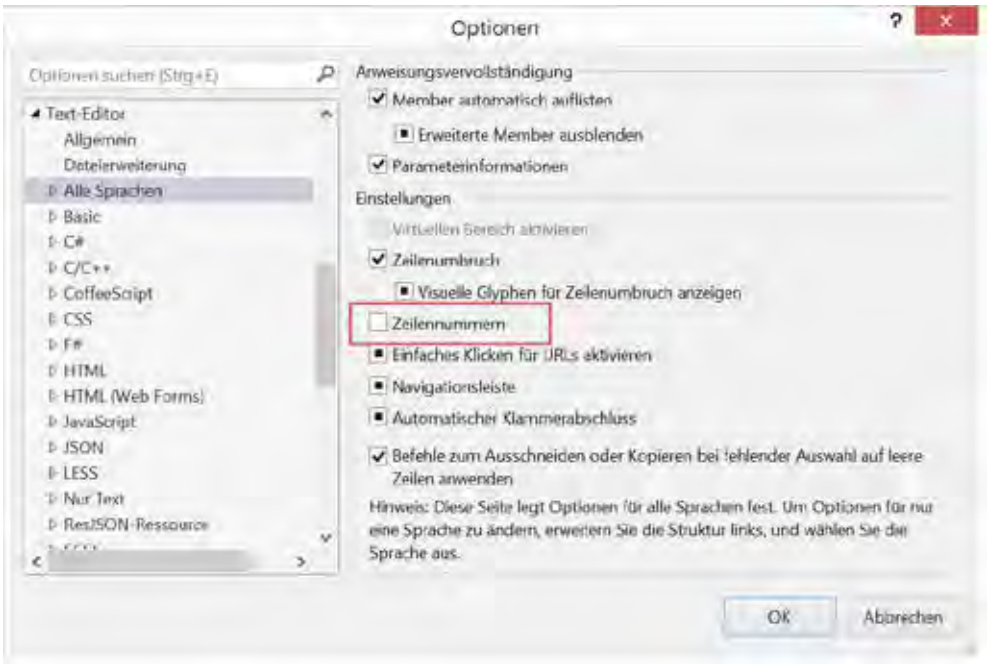


Bild 4: Einblenden der Zeilennummern

die Zeilennummern anzeigt. Im Screenshot ist dies nicht der Fall, aber Sie blenden die Zeilennummern ganz einfach mit der Option **Zeilennummern** in den Visual Studio-Optionen (**Extras|Optionen**) im Bereich **Text-Editor|Alle Sprachen** ein (siehe Bild 4).

Im vorliegenden Fall beheben Sie die knapp zehn Fehler übrigens, indem Sie den Zeilenumbruch innerhalb der Zeichenkette wieder entfernen.

Groß und klein

Wenn Sie schnell einen Syntaxfehler provozieren möchten, brauchen Sie nur einmal **WriteLine** mit ausschließlich kleinen Buchstaben zu schreiben. Visual Studio korrigiert dies nicht – zumindest dann nicht, wenn Sie die IntelliSense-Option nicht nutzen.

Stattdessen zeigt die Fehlerliste den Fehler **"System.Console" enthält keine Definition für "writeline".** an.

Die Groß- und Kleinschreibung ist also wichtig – Visual Studio erkennt Elemente der Objektbibliothek nicht, wenn diese nicht hundertprozentig richtig geschrieben sind.

Dies gilt übrigens auch für die Benennung von Objekten, Variablen et cetera: Sie können durchaus zwei Variablen namens **kundeid** und **KundeID** verwenden. Durch die unterschiedliche Groß- und Kleinschreibung werden diese als verschiedene Variablen erkannt.

Blockweise

C#-Code ist, bis auf die oberste Ebene, immer in Blöcke eingefasst. Ein Block wird durch eine öffnende geschweifte Klammer vor dem Block und eine schließende geschweifte Klammer hinter dem Block gekennzeichnet. Blöcke stehen meist in einem bestimmten Kontext, beispielsweise als Inhalt einer Methode, eines Namespaces, einer Schleife oder einer Bedingung.

Sie können aber auch einfach so einen Block mit geschweiften Klammern zusammenfassen. Hier ein sinnloses Beispiel, dass aber zeigt, wie es aussehen kann:

```
static void Main(string[] args)
{
    Console.WriteLine("Eingabetaste drücken!");
    {
        Console.ReadLine();
    }
}
```

Kommentare

Einzeilige Kommentare leiten Sie mit zwei Schrägstrichen ein:

```
//Kommentarzeile
```

Im Gegensatz zu VBA können Sie auch mehrere Zeilen zu einem Kommentarbereich zusammenfassen:

```
/*Mehrzeiliger
Kommentar*/
```

Wie unter VBA können Sie auch einen Kommentar hinter einer Zeile anbringen:

```
Console.WriteLine("Blabla"); //
Kommentar
```

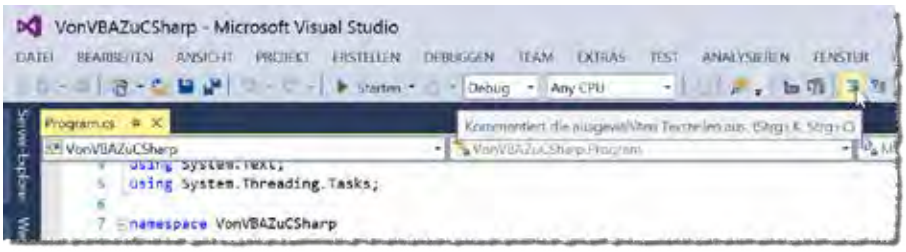


Bild 5: Symbolleistenbefehl zum Auskommentieren von Codezeilen

Schnell auskommentieren

Wenn Sie unter VBA schnell einmal ein paar Zeilen gleichzeitig auskommentieren wollten, haben Sie die Symbolleiste **Bearbeiten** ein-

Von VBA zu C#: Operatoren

Die Anzahl der Operatoren unter VBA ist überschaubar. Die .NET-Programmiersprachen bieten hier schon eine ganze Menge neuer Möglichkeiten. Zum Beispiel brauchen Sie, wenn Sie einen Zähler um eins erhöhen wollen, nicht mehr x = x + 1 zu schreiben, sondern einfach nur x += 1. Diese und viele andere Operatoren stellt der vorliegende Artikel vor.

Operatoren

Es gibt verschiedene Arten von Operatoren, die wir in die folgenden Kategorien einteilen:

- Zuweisungsoperatoren
- Arithmetische Operatoren
- Vergleichsoperatoren
- Logische Operatoren
- Andere Operatoren

Beispiele

Wenn Sie die nachfolgend vorgestellten Beispiele ausprobieren möchten, erstellen Sie einfach eine C#-Konsolenanwendung. Fügen Sie die Anweisungen in die Methode **static void Main()** ein und führen Sie diese durch Betätigen der Taste **F5** aus.

Zuweisungsoperatoren

Als Erstes schauen wir uns die Operatoren an, mit denen Sie einer Variablen einen neuen Wert zuweisen können:

- **x = y**: **x** erhält den Wert von **y**.
- **x += y**: **y** wird zu **x** addiert und in **x** gespeichert (entspricht unter VBA dem Ausdruck **x = x + y**)
- **x -= y**: **y** wird von **x** subtrahiert (**x = x - y**)
- **x *= y**: **x** wird mit **y** multipliziert (**x = x * y**)
- **x /= y**: **x** wird durch **y** dividiert (**x = x / y**)
- **x %= y**: Liefert den Rest von **x/y** (**x = x mod y**)
- **x &= y**: Bitweise AND-Operation (mehr weiter unten – unter VBA **AND**)
- **x |= y**: Bitweise OR-Operator (mehr weiter unten – unter VBA **OR**)
- **x ^= y**: XOR-Operator (mehr weiter unten)
- **x <<= y**: Verschiebt **x** um **y** bitweise nach links.
- **x >>= y**: Verschiebt **x** um **y** bitweise nach rechts.
- **=>**: Lambda-Operator. Darauf gehen wir später ein.

Bitweise Vergleichsoperationen

Die Auflistung oben enthielt einige bitbezogene Operatoren. Diese vergleichen die einzelnen Bits von Zahlenwerten und liefern das entsprechende Ergebnis. Wenn Sie also beispielsweise die Zahl **6** (binär **110**) und **2** (**10**) vergleichen, erhalten Sie bei der **AND**-Operation den Wert **2** (also **10**), weil nur das zweite Bit bei beiden Zahlenwerten gesetzt ist.

Bei einer **OR**-Operation mit den Zahlen **4** (**100**) und **1** (**1**) käme der Wert **5** heraus (**101**).

Die **XOR**-Operation liefert nur solche Bits, die sich bei beiden Operatoren unterscheiden. Der Vergleich von **7** (**111**) und **3** (**11**) würde also **4** ergeben (**100**).

Die bitweise Verschiebung mit den Operatoren **<<=** und **>>=** sorgt dafür, dass die Bits um die angegebene Anzahl nach links oder rechts verschoben werden:

```
Console.WriteLine(5 << 1);
//Liefert: 10
```

5 entspricht binär dem Wert **101**. Wenn Sie **101** um eine Position nach links verschieben, wird der freie Platz rechts mit dem Wert **0** aufgefüllt, Sie erhalten also **1010**, was dem dezimalen Wert **10** entspricht.

Verschieben Sie **101** hingegen um eins nach rechts, erhalten Sie **10** – aus **5** wird so also **2**:

```
Console.WriteLine(5 >> 1);
//Liefert: 2
```

Arithmetische Operatoren

Unter den arithmetischen Operatoren verstehen wir die Grundrechenarten sowie einige weitere. Die Operatoren für Addition (**+**), Subtraktion (**-**), Multiplikation (*****) und Division (**/**) brauchen wir nicht weiter zu erläutern.

Das Plus- und das Minuszeichen dienen außerdem als Vorzeichen.

Daneben gibt es noch das Prozentzeichen (**%**), mit dem Sie den Rest einer Division ermitteln:

```
x = 5;
y = 2;
Console.WriteLine("Der Rest von 5/2 ist: {0}", y % x);
//Liefert 1
```

Beim Dividieren zweier Zahlen vom Datentyp **int** müssen Sie beachten, dass das Ergebnis wieder den gleichen Datentyp hat. Im folgenden Beispiel wird aus **5/2** dann **2** statt des erwarteten Wertes **2.5**:

```
int x = 5;
int y = 2;
Console.WriteLine("5/2 ist: {0}", y / x);
//Liefert 2
```

Der Grund ist, dass für das Ergebnis einer Rechenoperation der größte Datentyp verwendet wird – und der ist in diesem Fall **int**. Dementsprechend gibt die Konsole den Wert **2** statt **2.5** aus.

Wenn Sie mindestens einen der beiden Werte als Datentyp mit Nachkommastellen deklarieren, erhalten Sie das gewünschte Ergebnis:

```
int e = 5;
double f = 2;
Console.WriteLine("5/2 ist: {0}", e/f);
//Liefert 2.5
```

Interessanterweise tritt das gleiche Problem auf, wenn Sie ohne Variablen arbeiten und die Gleichung mit konkreten Zahlenwerten angeben:

```
Console.WriteLine("5/2 ist: {0}", 5 / (double)2);
//Liefert 2
```

Dies können Sie ändern, indem Sie einen der Werte explizit als **double** deklarieren. Dies erledigen Sie durch Voranstellen des Schlüsselworts **(double)** in Klammern:

```
Console.WriteLine("5/(double)2 ist: {0}", 5 / (double)2);
//Liefert 2.5
```

Alternativ geben Sie eine echte Dezimalzahl an, also mit Nachkommastelle:

```
Console.WriteLine("5/2.0 ist {0}", 5 / 2.0);
//Liefert 2.5
```

Inkrement und Dekrement

Der Operator **++** vor einer Zahl oder einer Variablen addiert **1** zum Wert hinzu (Inkrementoperator):

```
x = 5;
Console.WriteLine("++5 ist: {0}", ++x);
//Ergibt 6
```

-- hingegen subtrahiert den Wert 1 (Dekrement-operator):

```
x = 5;
Console.WriteLine("--5 ist: {0}", --x);
//Ergibt 4
```

Vergleichsoperatoren

Die Vergleichsoperatoren kleiner (<), größer (>), kleiner gleich (<=), größer gleich (>=) und gleich (==) bedürfen keiner weiteren Erläuterung. Wichtig für Umsteiger von VBA ist hier, dass für die Prüfung der Gleichheit ein doppeltes Gleichheitszeichen und kein einfaches verwendet wird. Sie nutzen also nicht etwa diesen Ausdruck:

```
Debug.Print 1=2
False
```

Stattdessen nutzen Sie das doppelte Gleichheitszeichen:

```
Console.WriteLine(1 == 2);
//Liefert False
```

Das einfache Gleichheitszeichen würde als Zuweisung interpretiert werden, was aber in diesem Fall zu einem Fehler führt, weil man einer Zahl keinen Wert zuweisen kann. Auch der unter VBA mit dem Zeichen <> abgebildete Ungleich-Operator sieht unter C# anders aus. Hier verwenden Sie den Operator !=:

```
Console.WriteLine(1 != 1);
//Liefert False
```

Für Objekte ist noch der is-Vergleichsoperator interessant. Diesen erläutern wir später in einem Artikel über die Grundlagen der Objektorientierung.

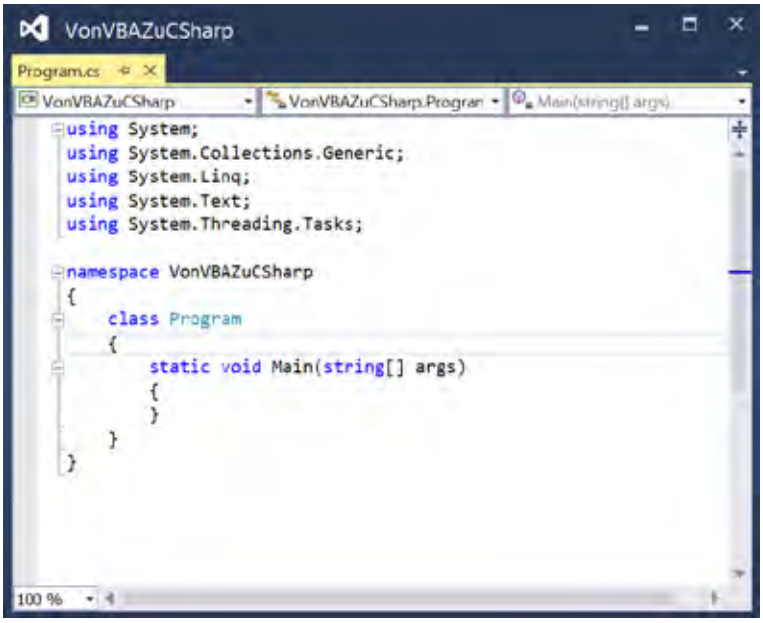


Bild 1: Standardmäßig vorhandenes Klassenmodul in einem neuen C#-Projekt

Logische Operatoren

Der UND-Operator (&) kann für Boolean- und ganzzahlige Werte verwendet werden. Bei Boolean-Vergleichen liefert dies folgende Ergebnisse:

```
Console.WriteLine(false & false);
//Liefert False
Console.WriteLine(false & true);
//Liefert False
Console.WriteLine(true & false);
//Liefert False
Console.WriteLine(true & true);
//Liefert True
```

Bei Ganzzahlen erfolgt mit dem UND-Operator ein binärer Vergleich. Kurz gefasst werden dabei die einzelnen Bits verglichen und das Ergebnis zu einer neuen Zahl zusammengefasst. Beispiel:

```
Console.WriteLine("5 und 3 ist {0}", 5 & 3);
//Liefert 1
```

5 ist binär 101, 3 ist binär 011. Es werden jeweils die einzelnen Stellen verglichen, also 1 und 0, 0 und 1 und 1

Von VBA zu C#: Bedingungen

Unter Access-VBA haben Sie als Bedingungen die Befehlsstrukturen If...Then und deren Varianten sowie Select Case kennen gelernt. Außerdem gibt es noch Funktionen, mit denen sich bestimmte Werte in Abhängigkeit des Wertes eines Parameters zurückgeben lassen – zum Beispiel IIf oder Choose. Unter C# sieht dies etwas anders aus, vor allem wegen der Strukturierung mit geschweiften Klammern und des fehlenden End-Schlüsselwortes. Dieser Beitrag stellt die Pendants zu den VBA-Bedingungen vor.

Einfaches If...Then

Die übliche **If...Then**-Bedingung bauen Sie in VBA wie folgt auf:

```
If intZahl = 0 Then
    MsgBox "Die Zahl lautet Null."
End If
```

Unter C# können Sie diese Bedingung sogar noch einfacher schreiben:

```
if (Zahl == 0)
    Console.WriteLine("Die Zahl lautet Null.");
```

Wenn Sie diese Anweisungen ausprobieren möchten, erstellen Sie eine neue C#-Konsolenanwendung mit Visual Studio und ergänzen die Main-Prozedur wie folgt:

```
static void Main(string[] args) {
    int Zahl;
    Console.WriteLine("Geben Sie eine ganze Zahl ein.");
    Zahl = Convert.ToInt32( Console.ReadLine());
    if (Zahl == 0)
        Console.WriteLine("Die Zahl lautet Null.");
}
```

Das Ergebnis sieht dann etwa wie in Bild 1 aus. Wieso aber gelingt dies mit nur zwei Zeilen – und auch noch ohne geschweifte Klammern? Nun: Dies ist eine vereinfachte Form, bei der Sie nur eine einzige Zeile eingeben dürfen. Sobald Sie mehr als eine Zeile verwenden wollen, müssen Sie die Anweisungen in geschweifte Klammern einfassen. Dies können Sie der Konsistenz halber allerdings auch bereits bei Verwendung einer einzigen Anweisung tun:

```
if (Zahl == 0) {
    Console.WriteLine("Die Zahl lautet Null.");
}
```

Die **If...Then**-Bedingung gerät unter C# auf jeden Fall zur **If**-Bedingung, denn das Schlüsselwort **Then** wird ja hier gar nicht benötigt.

If...Then...Else-Bedingung

Nehmen wir noch einen **Else**-Zweig hinzu. Unter VBA verwenden Sie folgende Struktur:

```
If intZahl = 0 Then
    MsgBox "Die Zahl lautet Null."
Else
    MsgBox "Die Zahl lautet nicht Null."
End If
```

Unter C# könnten Sie hier wieder die Syntax für die Einzeiler verwenden:

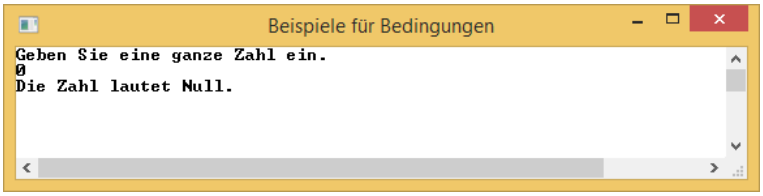


Bild 1: Ausgabe eines Ergebnisses per **If**-Bedingung

```
if (Zahl == 0)
    Console.WriteLine("Die Zahl lautet Null.");
else
    Console.WriteLine("Die Zahl lautet nicht Null.");
```

Mit geschweiften Klammern erhalten Sie folgenden Code. Hier können Sie nun auch noch weitere Anweisungen je Zeile unterbringen:

```
if (Zahl == 0) {
    Console.WriteLine("Die Zahl lautet Null.");
}
else {
    Console.WriteLine("Die Zahl lautet nicht Null.");
}
```

Verschachteltes If...Then...Else

Wenn Sie zwei **If...Then**-Bedingungen unter VBA verschachteln wollten, haben Sie dies einfach nach folgendem Schema erledigt:

```
If <Äußere Bedingung> Then
    If <Innere Bedingung> Then
        MsgBox "Äußere wahr, innere wahr"
    Else
        MsgBox "Äußere wahr, innere falsch"
    End If
End If
```

Wenn Sie unter C# ohne geschweifte Klammern arbeiten, lässt sich die Zuordnung des **else**-Teils einer verschachtelten **if...else**-Konstruktion möglicherweise nicht sehr gut erkennen. Einfacher ist es immer, mit geschweiften Klammern zu arbeiten – zumindest aus Gründen der Übersicht. Dies sieht dann etwa wie folgt aus:

```
if (Zahl != 0) {
    if (Zahl > 5) {
        Console.WriteLine("Die Zahl ist größer als 5.");
    }
    else {
        Console.WriteLine("Die Zahl ist <= 5.");
    }
}
```

In diesem Fall prüft die äußere **if**-Bedingung, ob der Benutzer eine Zahl größer **0** eingegeben hat. Die innere prüft, ob die Zahl größer als **5** ist und gibt eine entsprechende Meldung aus. Beim Wert **0** oder negativen Zahlen geschieht nichts.

If...ElseIf

Es fehlt noch die von VBA bekannte Möglichkeit, mehr als nur den **If**- und den **Else**-Teil auszuwerten – nämlich der **ElseIf**-Zweig (oder auch mehrere):

```
If <Erste Bedingung> Then
    MsgBox "Erste Bedingung"
ElseIf <Zweite Bedingung> Then
    MsgBox "Zweite Bedingung"
Else
    MsgBox "Andere Bedingung"
End If
```

Ein **ElseIf** gibt es unter C# nicht. Hier müssen Sie sich mit immer weiter verschachtelten **else**/**if**-Bedingungen behelfen.

Sprich: Wenn die erste **if**-Bedingung nicht erfüllt wird (**Zahl == 0**), wird der **else**-Teil angelaufen, der eine weitere **if**-Bedingung enthält (**Zahl == 1**) – und einen weiteren **else**-Teil:

```
if (Zahl == 0) {
    Console.WriteLine("Die Zahl lautet 0.");
}
else {
    if (Zahl == 1) {
        Console.WriteLine("Die Zahl lautet 1.");
    }
    else {
        if (Zahl == 2) {
```

Von VBA zu C#: Schleifen

Schleifen spielen bei der Programmierung eine wichtige Rolle. Unter VBA gibt es die For...Next- und die For...Each-Schleife sowie die verschiedenen Varianten der Do...-Schleife. In diesem Artikel schauen wir uns an, wie Sie diese Schleifentypen unter C# abbilden und welche Besonderheiten dabei auftreten.

Beispiele

Wenn Sie die Beispiele dieses Artikels ausprobieren möchten, erstellen Sie eine C#-Konsolenanwendung und fügen die Anweisungen in die Prozedur **static void main()** ein. Betätigen Sie dann die Taste **F5**, um die Methode zu starten.

For...Next-Schleife

Die **For...Next**-Schleife aus VBA sieht so aus:

```
For i = 1 To 10
    Debug.Print "Aktuelle Zahl: " & i
Next i
```

Unter C# sieht dies etwas anders aus, lässt sich aber identisch abbilden:

```
for (int i = 1; i <= 10; i++)
{
    Console.WriteLine("Aktuelle Zahl: {0}", i);
}
```

Der wichtigste Teil ist hier die erste Zeile – die innerhalb der Schleife ausgeführten Anweisungen packen Sie, wie von C# gewohnt, einfach in geschweifte Klammern. Die erste Zeile enthält das Schlüsselwort **for** sowie in Klammern drei Parameter:

- Der erste deklariert die Laufvariable, hier **i**, und den Startwert der Variablen.
- Der zweite legt die Abbruchbedingung fest. In diesem Fall soll die Schleife solange laufen, wie **i** einen Wert kleiner oder gleich **10** aufweist. Die Abbruchbedingung muss immer einen Boolean-Wert als Ergebnis liefern.
- Schließlich folgt der dritte Parameter, der die Änderung der Laufvariablen festlegt. In diesem Fall sorgt **i++** dafür, dass **i** mit jedem Durchlauf um den Wert **1** erhöht wird.

Das Ergebnis dieser Schleife sieht so aus:

```
Aktuelle Zahl: 1
Aktuelle Zahl: 2
Aktuelle Zahl: 3
...
Aktuelle Zahl: 10
```

Deklaration der Laufvariablen

Nun haben wir die Laufvariable direkt im ersten Parameter der Schleife deklariert und instanziert:

```
int i = 1
```

Dadurch legen wir gleichzeitig den Gültigkeitsbereich der Variablen auf die Schleife selbst fest. Nach dem Beenden der Schleife können Sie nicht mehr auf die Variable zugreifen.

Das ist aber kein Problem, denn für Fälle, in denen Sie den Inhalt der Variablen später noch benötigen, deklarieren Sie die Laufvariable einfach vor Beginn der Schleife. Dies sieht dann wie folgt aus:

```
int i;
for (i = 1; i <= 10; i++)
{
    Console.WriteLine("Aktuelle Zahl: {0}", i);
}
Console.WriteLine("Wert von i: {0}", i);
```

Die abschließende Anweisung gibt nun den Wert der Zählervariablen nach dem letzten Durchlaufen der Schleife aus:

```
Aktuelle Zahl: 1
Aktuelle Zahl: 2
Aktuelle Zahl: 3
...
Aktuelle Zahl: 10
Wert von i: 11
```

Hätten Sie im ersten Beispiel auf die gleiche Art auf die Laufvariable zugegriffen, hätte dies einen Syntaxfehler hervorgerufen.

Ob Sie die Zählervariablen nun jeweils vor dem Start der Schleife oder erst in der **for...**-Anweisung deklarieren, hängt von der geplanten Nutzung dieser Variable ab – soll diese vor dem Start der Schleife oder im Anschluss gefüllt oder ausgelesen werden, müssen Sie die Deklaration natürlich vor dem ersten Zugriff programmieren. Anderenfalls haben Sie die freie Wahl.

Herunterzählen

Wenn Sie der Wert der Zählervariablen beim Initialisieren der Schleife größer sein soll als beim Beenden, legen Sie die Werte im ersten und zweiten Parameter der **for...**-Anweisung entsprechend fest. Für den dritten Parameter definieren Sie dann die Schrittweiter für das Herunterzählen. Angenommen, Sie wollen jeweils den Wert **1** abziehen, verwenden Sie an dieser Stelle den Ausdruck **i--**:

```
int i;
for (i = 10; i >= 1; i--)
{
    Console.WriteLine("Aktuelle Zahl: {0}", i);
}
```

Schrittweite festlegen

Was geschieht nun, wenn Sie der Zählervariablen nicht mit jedem Durchlauf genau den Wert **1** hinzufügen möchten, sondern beispielsweise den Wert **2**? Die Antwort ist einfach: Sie können jeden beliebigen Ausdruck für den dritten Parameter verwenden, welcher der Laufvariablen einen Wert zuweist. Im folgenden Beispiel sollen alle Werte von **0** bis **10** mit der Schrittweite **2** durchlaufen werden. Also verwenden wir für den dritten Parameter den Ausdruck **i +=2**:

```
int i;
for (i = 0; i <= 10; i += 2)
{
    Console.WriteLine("Aktuelle Zahl: {0}", i);
}
```

Sie könnten dort auch nach guter, alter VBA-Manier **i = i + 2** einsetzen:

```
for (i = 0; i <= 10; i = i + 2)
```

Wichtig ist allein, dass Sie **i** dort einen Wert zuweisen.

Verschachtelte for-Schleifen

Genau wie die **For...Next**-Schleife unter VBA können Sie auch die **for**-Schleife unter C# verschachteln.

Das folgende Beispiel soll beispielsweise eine äußere Schleife mit der Laufvariablen **x** und eine innere Schleife mit der Laufvariablen **y** durchlaufen. Dabei soll die Anweisungen der inneren Schleife das Produkt der aktuellen Werte von **x** und **y** hintereinander ausgeben, wobei jeweils fünf Zeichen Platz für die Zahlenwerte vorgesehen werden sollen.

Beim ersten Durchlauf der äußeren Schleife geschieht zunächst nichts, dann beginnt der erste Durchlauf der inneren Schleife mit der Ausgabe des Produkts von **1** und **1**. In der Folge wird die innere Schleife solange durchlaufen, bis die Abbruchbedingung erfüllt ist. Danach folgt die letzte Anweisung der äußeren Schleife, die dafür sorgt, dass die zehn in der aktuellen Zeile der Konsole eingetragenen Werte mit einem Zeilenumbruch abgeschlossen werden:

Von VBA Zu C#: Arrays

Arrays gibt es wohl in jeder Programmiersprache, jeweils mit eigenen Besonderheiten. Wie füllt man ein Arrays? Wie greift man auf die Werte eines Arrays zu? Wie organisiert man mehrdimensionale Arrays? Wie durchläuft man die Werte eines Arrays oder ermittelt die Anzahl der enthaltenen Elemente? All diese Fragen beantwortet der vorliegende Artikel.

Unter VBA konnten Arrays nur einfache Werttypen aufnehmen, also beispielsweise Zahlen oder Zeichenketten. Wenn Sie etwa Objekte in einer Liste speichern wollten, mussten Sie auf eine Alternative wie das **Collection**- oder das **Dictionary**-Objekt zugreifen.

Unter C# können Sie dies gleichermaßen mit einem Array erledigen. Dies ist jedoch nicht der einzige Unterschied. In den folgenden Abschnitten erhalten Sie die Grundlagen für die Programmierung von Arrays mit C#.

Deklaration von Arrays

Unter VBA haben Sie ein Array durch das Hinzufügen eines Klammernpaares bei der Deklaration kenntlich gemacht:

```
Dim Zahlwoerter() As String
```

Wenn Sie schon wussten, wie viele Elemente das Array enthalten sollte, haben Sie diese Anzahl in Klammern angegeben:

```
Dim Zahlwoerter(9) As String
```

Dies bedeutete ohne weitere Angaben, dass das Array zehn Elemente mit den Indizes **0** bis **9** aufnehmen konnte.

Unter C# verwenden Sie keine runden, sondern eckige Klammern, und diese geben Sie auch nicht hinter dem Variablennamen, sondern hinter dem Datentyp an:

```
string[] Zahlwoerter;
```

Nun ist natürlich noch nicht bekannt, wie viele Elemente das Array aufnehmen soll. Dies geben Sie auch nicht bekannt, indem Sie die Anzahl einfach in die eckigen Klammern schreiben. Vielmehr ist eine Initialisierung unter Angabe der Anzahl der Elemente nötig. Diese können Sie in einer neuen Zeile erledigen:

```
Zahlwoerter = new string[10];
```

Oder Sie verwenden einfach einen Einzeiler zur gleichzeitigen Deklaration und Initialisierung:

```
string[] Zahlwoerter = new string[10];
```

Die Verwendung des Schlüsselworts **new** stellt hier im Gegensatz zum Vorgehen unter VBA heraus, dass es sich bei dem Array um ein Objekt handelt.

Stellt sich gleich die erste Frage: Besitzt das Array nun zehn Elemente? Und die zweite gleich hinterher: Wie werden diese indiziert – beginnend mit **0** oder **1**?

Das Array enthält nun tatsächlich genau die angegebene Elementanzahl. Im Gegensatz zu der etwas unübersichtlichen Vorgehensweise von VBA, bei der die Anzahl von der Basis des Index abhängt, initialisiert C# genau mit der angegebenen Menge von Elementen.

Der Index basiert unter C# auf dem Wert **0**. Auf ein Element greifen Sie über die Angabe des Indexwertes in eckigen Klammern zu, also etwa über **Zahlwoerter[0]** – mehr dazu weiter unten.

Array füllen

Schauen wir uns erstmal eine einfache Methode an, das Array zu füllen – und zwar direkt bei der Initialisierung.

Ausgehend vom obigen Einzeiler gelingt dies so (in einer Zeile):

```
string[] Zahlwoerter = new String[10] { "Eins", "Zwei",
"Drei", "Vier", "Fünf", "Sechs", "Sieben", "Acht", "Neun",
"Zehn" };
```

Sie geben also einfach die gewünschten Werte in einer geschweiften Klammer und durch Kommata getrennt an.

In diesem Fall soll das Array zehn Elemente aufnehmen und jedes der Elemente wird auch gleich gefüllt.

Es gibt noch eine Alternative, bei der Sie die Anzahl der Elemente nicht angeben und diese direkt aus der Anzahl der übergebenen Elemente ermittelt wird. In diesem Fall lassen Sie einfach die Angabe der Anzahl innerhalb der eckigen Klammern weg (in einer Zeile):

```
string[] Zahlwoerter = new String[] { "Eins", "Zwei",
"Drei", "Vier", "Fünf", "Sechs", "Sieben", "Acht", "Neun",
"Zehn" };
```

Anschließend geben wir die im Array gespeicherten Werte in der Konsole aus. Für den Zugriff auf den Inhalt eines der Elemente geben Sie den Variablennamen gefolgt vom gewünschten Index in eckigen Klammern an.

Am einfachsten bekommen wir den Inhalt eines Elements so auf die Konsole:

```
Console.WriteLine(Zahlen[0]);
```

Wir wollen aber gleich alle zehn, und dann auch noch mit einem kleinen Hinweistext:

```
Console.WriteLine("Wert von Zahlwoerter[0]: {0}", Zahlen[0]);
Console.WriteLine("Wert von Zahlwoerter[1]: {0}", Zahlen[1]);
Console.WriteLine("Wert von Zahlwoerter[2]: {0}", Zahlen[2]);
...
Console.WriteLine("Wert von Zahlwoerter[9]: {0}", Zahlen[9]);
```

Das Ergebnis sieht nun wie folgt aus:

```
Wert von Zahlwoerter[0]: Eins
Wert von Zahlwoerter[1]: Zwei
Wert von Zahlwoerter[2]: Drei
...
Wert von Zahlwoerter[9]: Zehn
```

Wichtig: Wenn Sie die Anzahl der zu erstellenden Elemente angeben und gleich dahinter die Elemente hinzufügen, muss deren Anzahl übereinstimmen.

Die schnellste Alternative, um ein Array direkt mit Werten zu füllen, lässt noch ein paar Schlüsselwörter weg und übergibt die Werte direkt als Array an die Variable:

```
string[] Zahlwoerter = { "Eins", "Zwei", "Drei", "Vier",
"Fünf", "Sechs", "Sieben", "Acht", "Neun", "Zehn" };
```

Elemente nach Initialisierung füllen

Wenn Sie das Array zunächst deklarieren und initialisieren, aber noch nicht füllen, holen Sie dies in späteren Anweisungen nach.

Dazu weisen Sie einfach dem entsprechenden Element, das Sie über die Angabe des Indexwertes in eckigen Klammern referenzieren, den gewünschten Wert zu:

```
string[] Zahlwoerter = new string[10];
Zahlwoerter[0] = "Eins";
Zahlwoerter[1] = "Zwei";
Zahlwoerter[2] = "Drei";
```

Wenn Sie nun alle Elemente ausgeben, bleiben die hinteren sieben leer. Die Elemente werden bei der Initialisierung gleich mit leeren Zeichenketten gefüllt.

0 als initialer Wert von Zahlen-Arrays

Probieren wir dies gleich noch mit Zahlen aus, um zu prüfen, mit welchem Wert das Element eines Zahlen-Arrays initial gefüllt wird:

```
int[] Zahlen = new int[5];
Zahlen[0] = 1;
Zahlen[1] = 2;
Zahlen[2] = 3;
Console.WriteLine("Wert von Zahlen[0]: {0}", Zahlen[0]);
Console.WriteLine("Wert von Zahlen[1]: {0}", Zahlen[1]);
Console.WriteLine("Wert von Zahlen[2]: {0}", Zahlen[2]);
Console.WriteLine("Wert von Zahlen[3]: {0}", Zahlen[3]);
Console.WriteLine("Wert von Zahlen[4]: {0}", Zahlen[4]);
```

Wir definieren also ein Array mit fünf **int**-Elementen und füllen dann nur drei davon. Das Ergebnis bei der Ausgabe aller Elemente inklusive der übrigen zwei sieht so aus:

```
Wert von Zahlen[0]: 1
Wert von Zahlen[1]: 2
Wert von Zahlen[2]: 3
Wert von Zahlen[3]: 0
Wert von Zahlen[4]: 0
```

Leere Zahlenelemente werden also direkt mit dem Wert **0** gefüllt.

Größe einer Dimension eines Array ermitteln

Unter VBA ermitteln Sie die Größe eines Arrays über die Differenz der Funktionen **UBound** und **LBound** mit dem Array als Parameter:

```
Debug.Print UBound(intZahlen()) - LBound(intZahlen())
```

Für mehrdimensionale Arrays gibt es noch weitere Besonderheiten. Unter C# gibt es verschiedene Eigenschaften, mit denen Sie die Anzahl der Elemente einer Dimension sowie die Gesamtzahl der Elemente ermitteln können. Zunächst reicht uns die Anzahl der Elemente einer einzigen Dimension. Die erhalten Sie mit der Funktion **GetLength**:

```
Console.WriteLine(Zahlen.GetLength(0));
```

GetLength erwartet als Parameter den Index der zu untersuchenden Dimension, in diesem Fall **0**.

Im Falle eines eindimensionalen Arrays liefert die Funktion **Length** den identischen Wert. Für mehrdimensionale Arrays liefert sie hingegen die Anzahl aller Elemente. Dementsprechend kommt sie ohne Parameter aus:

```
Console.WriteLine(Zahlen.Length);
```

Größe des Arrays zur Laufzeit ändern

Weiter oben haben Sie bereits gesehen, dass Sie die Größe eines Arrays zur Laufzeit festlegen können. Dies ist gleichbedeutend mit der Initialisierung. Den dort angegebenen Zahlenwert, der die Anzahl der Elemente angibt, kann auch als Variable übergeben werden – nicht immer ist ja gleich beim Programmieren bekannt, wie groß ein Array gegebenenfalls werden soll.

Im folgenden Beispiel legen wir die Anzahl der Elemente per Variable fest (die natürlich auch etwa über die Konsole abgefragt oder auf andere Weise ermittelt werden kann) und initialisieren das Array damit:

```
string[] Zahlwoerter;
int AnzahlZahlwoerter;
AnzahlZahlwoerter = 10;
Zahlwoerter = new string[AnzahlZahlwoerter];
```

Aber was geschieht, wenn wir das Array bereits mit Werten füllen oder auslesen wollen, aber später die Größe des Arrays ändern müssen? Mit der Initialisierung per **new**-Schlüsselwort ist uns nicht geholfen, da das Array dann ja wieder leer ist.

Genau genommen gibt es bei den Arrays keine Möglichkeit, es ohne Umweg zu vergrößern oder zu verkleinern. Immerhin gibt es einen Workaround. Dabei verwenden wir neben dem eigentlichen Array **Zahlen** ein Hilfsarray namens **temp**. **Zahlen** deklarieren wir als Array mit der Elementanzahl **0**, **temp** zunächst ohne Initialisierung.

Dann durchlaufen wir eine Schleife von **1** bis **10**, da wir dem Array mit jedem Durchlauf ein weiteres Element

Die Console-Klasse

Im Gegensatz zu VBA können Sie mit den .NET-Programmiersprachen wesentlich komfortabler auf die Eingabeaufforderung zugreifen – und zwar über die Console-Klasse des System-Namespaces. Dieses bietet nicht nur die Möglichkeit, Texte auszugeben, sondern auch solche zum Einlesen von Benutzereingaben. Außerdem können Sie die Eingabeaufforderung damit nach Ihren eigenen Wünschen formatieren. Dieser Artikel beschreibt die Verwendung der Console-Klasse.

Lesen und Schreiben

In einigen Beispielen anderer Artikel verwenden wir die Konsole, um ermittelte Daten auszugeben oder um mit dem Benutzer zu interagieren. Dafür benötigen wir vor allem die **Read...**- und **Write...**-Methoden der **Console**-Klasse. Wenn Sie die folgenden Beispiele ausprobieren möchten, legen Sie am einfachsten eine neue Konsolenanwendung für C# an. Dort fügen Sie die folgenden Beispielanweisungen einfach in die Prozedur **static void Main(string{] args)** ein und drücken auf **F5**, um diese auszuprobieren. Die **Write**-Methode gibt einfach den als Parameter angegebenen Text in der Konsole aus:

```
Console.Write("1");
```

Dummerweise wird die Konsole unmittelbar darauf wieder geschlossen, sodass diese bestenfalls kurz aufflackert.

Damit dies nicht geschieht, müssen Sie zumindest die Betätigung einer einzigen Taste einfordern, was der nachfolgende Aufruf der **ReadKey**-Methode bewirkt:

```
Console.Write("1");
Console.ReadKey();
```

Wenn Sie mehrere Ausgaben in einer Zeile tätigen wollen, beispielsweise in einer Schleife, rufen Sie einfach mehrmals die **Write**-Methode auf:

```
Console.Write("1");
Console.Write("2");
Console.ReadKey();
```

Das Ergebnis sieht dann wie in Bild 1 aus.

Auf Ihrem Rechner dürfte diese Meldung noch mit schwarzem Hintergrund und weißer Schrift erscheinen. Wir haben dies geändert, damit die Seiten dieses Artikels nicht so schwarz erscheinen – natürlich mit den Methoden und Eigenschaften des **Console**-Objekts. Dazu nutzen wir die Eigenschaften **BackgroundColor** und **ForegroundColor**. Beide sorgen dafür, dass die Hintergrund- und die Schriftfarbe für die geschriebenen Zeichen angepasst werden. Das bedeutet, dass der Rest des Fensters leider schwarz bleibt. Allerdings können wir dies ändern, indem wir nach dem Einstellen der beiden Eigenschaften noch die **Clear**-Methode des **Console**-Objekts auslösen:

```
Console.Title = "Konsole";
Console.BackgroundColor = ConsoleColor.White;
Console.ForegroundColor = ConsoleColor.Black;
```

Komplette Zeilen schreiben

Mit der **Write**-Methode schreiben Sie den angegebenen Text einfach in die Konsole und können später weitere Zei-

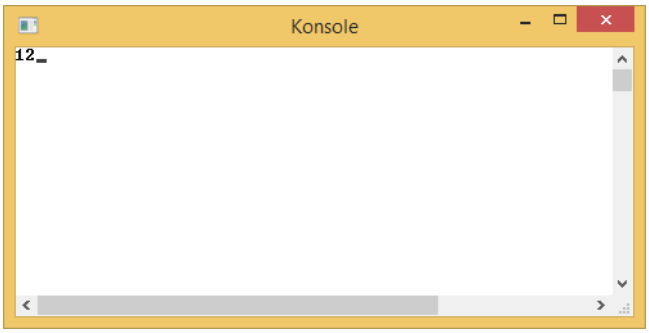


Bild 1: Ausgabe zweier **Write**-Anweisungen

chen an diesen Text anhängen. Wenn Sie nun direkt eine komplette Zeile inklusive Zeilenumbruch schreiben wollen, wie Sie es etwa von der **Debug.Print**-Methode unter VBA zum Schreiben von Zeilen in das Direktfenster kennen, verwenden Sie die Methode **WriteLine**:

```
Console.WriteLine("Eine Zeile.");
Console.ReadKey();
```

Als Ergebnis rutscht die Einfügemarke gleich in die folgende Zeile, weitere per **Write...** geschriebene Texte landen ebenfalls dort (siehe Bild 2).

Bild 2: Ausgabe einer kompletten Zeile mit **WriteLine**

Die **WriteLine**-Methode kommt in mehreren Varianten. Bei einer weiteren geben Sie mehrere Parameter an. Der erste enthält die eigentlich auszugebende Zeichenkette, die allerdings Platzhalter im Format **{0}**, **{1}** und so weiter enthalten kann. Die dafür einzusetzenden Werte geben Sie dann mit den nachfolgenden Parametern an, also beispielsweise so:

```
string Platzhalter = "Platzhalter";
Console.WriteLine("Text mit einem {0}.", Platzhalter);
Console.ReadKey();
```

Dies gibt die folgende Zeile aus:

```
Text mit einem Platzhalter.
```

Die Platzhalter können Sie auch noch durch einen zweiten Parameter erweitern, also etwa **{0, 5}**. Dies würde den für den ersten Platzhalter angegebenen Ausdruck mit einer Breite von mindestens fünf Zeichen rechtsbündig ausgeben:

```
string Platzhalter = "x.";
Console.WriteLine("Text mit einem {0, 5}", Platzhalter);
```

würde also folgendes ergeben:

```
Text mit einem     x.
```

Ist der Text für den Platzhalter länger als die reservierte Anzahl Zeichen, wird der vollständige Platzhaltertext ausgegeben. Für die linksbündige Ausgabe geben Sie einen negativen Wert als zweiten Parameter im Platzhalter an:

```
Console.WriteLine("Text mit einem {0,-5}.", Platzhalter);
```

liefert dann:

```
Text mit einem x    .
```

Es gibt noch eine Reihe weiterer Möglichkeiten, die Ausgabe zu formatieren. Dies würde jedoch den Rahmen dieses Artikels sprengen, daher gehen wir später in einem weiteren Artikel darauf ein.

Eingaben lesen

Es gibt drei Methoden, mit denen Benutzereingaben gelesen werden:

- **ReadKey**: Liefert ein Objekt des Typs **ConsoleKeyInfo** zurück, dass beispielsweise das gedrückte Zeichen und eventuell dabei betätigte Tasten wie **Strg**, **Alt** oder **Umschalt** liefert.
- **ReadLine**: Liest den Inhalt einer Zeile bis zum Betätigen eines Zeilenumbruchs ein und liefert diesen zurück.
- **Read**: Liefert den ASCII-Code des gedrückten Zeichens zurück.

Die ReadKey-Methode

Diese Methode wartet auf die Eingabe eines Zeichens und hält solange die aufrufende Routine an, bis der Benutzer

Von Access zu WPF: Fenster

Unter Access war alles so einfach und gewohnt. Das Access-Fenster bildete den Rahmen, zeigt das Ribbon an oder stellte die Formulare und Berichte in Anwendungen dar. Für das Öffnen der Objekte brauchte man nur die DoCmd.Open...-Anweisung zu kennen. Unter .NET sieht das ganz anders aus: Das Anwendungsfenster ist bereits ein Formular, es gibt andere Steuerelemente – und das Öffnen von Formularen erfolgt auch ganz anders.

Wenn Sie eine WPF-Anwendung erstellen (in diesem Beispiel für Programmiersprache C#), wählen Sie nach dem Starten von Visual Studio (hier in der Community-Edition der Version 2013) den Menüeintrag **Datei|Neu|Projekt...** aus. Danach erscheint der Dialog **Neues Projekt**, wo Sie unter **Vorlagen|Visual C#|Windows-Desktop** die Vorlage **WPF-Anwendung** selektieren (siehe Bild 1). Es gibt noch einige weitere Anwendungstypen auf Basis von WPF, aber wir konzentrieren uns hier zunächst auf die Desktop-Anwendung.

Visual Studio bietet hier direkt ein Verzeichnis an, in dem das Projekt standardmäßig angelegt wird. Diesen können Sie natürlich ändern. Wichtig ist zu wissen, dass im angegebenen Verzeichnis ein Unterverzeichnis mit dem Namen des Projekts angelegt wird, das schließlich die eigentlichen Projektdateien enthält. In diesem Fall soll das Projekt **VonAccessZuWPF** heißen.

Das Hauptfenster

Im Vergleich zu Access, wo das Access-Fenster das Hauptfenster der Anwendung ist (wenn Sie es nicht mit Tricks ausgeblendet haben), definieren Sie das Hauptfenster Ihrer WPF-Anwendung komplett neu. Dazu bietet Visual Studio im neuen Projekt standardmäßig ein Fenster namens **MainWindow** hinzu. Wir wollen uns an dieser Stelle daraufhin einigen, statt wie in Access von Formularen künftig von Fenstern zu sprechen.

Das Formular wird mithilfe von zwei Dateien beschrieben:

- **MainWindow.xaml**: Enthält die Beschreibung des Aussehens des Fensters.
- **MainWindow.xaml.cs**: Enthält den Code des Fensters. Unter Visual Basic würde die Endung **vb** lauten.

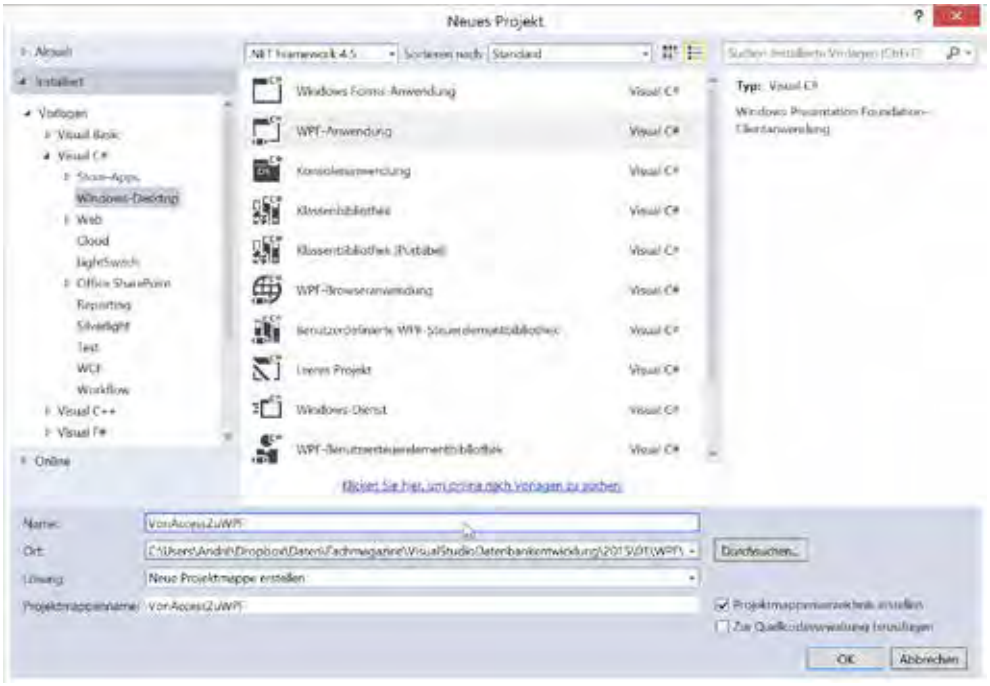


Bild 1: Neues WPF-Projekt für Visual C# erstellen

Die **.xaml**-Datei stellt Visual Studio in einem Bereich dar, der oben den Entwurf des Fensters anzeigt und unten den XML-Code, der das Aussehen definiert. Sie können das Aussehen auf folgenden Wegen beeinflussen:

- durch Ändern der Größe, Position oder anderer Eigenschaften des Fensters und der enthaltenen Steuerelemente direkt im Entwurf,
- durch Ändern der Eigenschaften im Bereich **Eigenschaften** oder
- durch Anpassen des XML-Codes zur Definition des Aussehens des Fensters.

Jegliche Änderungen an diesen drei Stellen wirken sich direkt auf die jeweils anderen Wege aus. Wenn Sie also im XML-Code die Breite des **Window**-Elements mit dem Attribut **Width** auf **600** einstellen, wird das Fenster direkt in dieser Breite dargestellt und auch der entsprechende Eintrag im Eigenschaftsfenster wird geändert.

Beispielprojekt: Den Code für die Beispiele der folgenden Abschnitte finden Sie in einem neuen, jungfräulichen WPF-Projekt.

Code des Hauptfensters

Die Datei **MainWindow.xaml** enthält den folgenden Code:

```
<Window x:Class="VonAccessZuWPF.MainWindow"
        xmlns="http://schemas.microsoft.com/winfx/2006/
            xaml/presentation"
        xmlns:x="http://schemas.microsoft.com/winfx/2006/xaml"
        Title="MainWindow" Height="350" Width="600">
    <Grid>
    </Grid>
</Window>
```

Hier finden Sie die beim Start vorhandenen Einstellungen: Der Titel lautet **MainWindow**, außerdem sind Höhe und Breite auf **350** und **600** festgelegt. Eine weitere wichtige Information finden Sie gleich in der ersten Zeile: Unter **x:Class** gibt das Dokument die C#-Datei an, die den Code für dieses Fenster enthält – in diesem Fall **VonAccessZuWPF.MainWindow**. Alle Elemente, die Sie nun zum Fenster hinzufügen, landen als Beschreibung innerhalb des **Grid**-Elements (**<Grid>...</Grid>**).

XAML und C#

Damit wird nun offensichtlich: Es gibt zwei verschiedene Dateien zur Programmierung eines Fensters unter WPF – die XML-Datei mit der XAML-Definition der Benutzeroberfläche und eine C#-Datei mit den Funktionen des Fensters.

Im Vergleich mit einem Access-Formular entspricht die Entwurfsansicht des WPF-Fensters dem Formularentwurf und die C#-Datei mit dem Code dem Klassenmodul des Formulars.

Unter Access werden intern allerdings sowohl die Beschreibung des Formulars sowie der Steuerelemente als auch der VBA-Code in einem einzigen Dokument untergebracht, was auch deutlich wird, wenn Sie dieses vom VBA-Fenster etwa mit der **SaveAsText**-Anweisung in eine Textdatei exportieren und in einem Texteditor ansehen.

Das »Klassenmodul« mit dem C#-Code hinter dem Fenster **MainWindow** hat den folgenden Inhalt:

```
namespace VonAccessZuWPF
{
    /// <summary>
    /// Interaktionslogik für MainWindow.xaml
    /// </summary>
    public partial class MainWindow : Window
    {
        public MainWindow()
        {
            InitializeComponent();
        }
    }
}
```

Hier geschieht noch nichts Spektakuläres, da wir noch keine Ereignisprozeduren et cetera angelegt haben. Die einzige Methode namens **InitializeComponent** erstellt kurz gefasst das Fenster auf Basis der XAML-Datei.

Wenn Sie genau wissen wollen, welcher Code sich hinter einer Anweisung befindet, klicken Sie ähnlich wie im VBA-Editor mit der rechten Maustaste und wählen den Eintrag **Definition einsehen** aus. Es erscheint dann ein gelb hinterlegter Bereich mit der Routine, die durch die Anweisung ausgelöst wird (siehe Bild 2).

Trennung von Oberfläche und Code

Die Definition der Benutzeroberfläche und der Code sind nicht nur dateiweise getrennt, sondern sie können auch komplett unabhängig voneinander bearbeitet werden. Während es unter Access kaum möglich ist, dass ein Entwickler Änderungen an der Benutzeroberfläche vornimmt, während ein anderer den Programmcode anpasst, gibt es für WPF mit **Blend für Visual Studio** sogar ein eigenes Werkzeug für die Verfeinerung der Benutzeroberfläche.

Vorteil Visual Studio: Zoom

Wenn Sie Ihr erstes Fenster erstellen, können Sie direkt ein Feature ausprobieren, das dem Access-Entwickler wohl für immer vorbehalten bleibt: Sie können die Ansicht des Entwurfs vergrößern oder verkleinern.

Dazu sollten Sie sich gleich die folgenden Tastenkombinationen merken:

- **Strg + Alt + +**: Vergrößern
- **Strg + Alt + -**: Verkleinern

Steuerelemente hinzufügen

Standardmäßig im linken Bereich von Visual Studio finden Sie den Werkzeugkasten. Dieser bietet verschiedene Sammlungen von Steuerelementen. Die obere liefert mit **Häufig verwendete WPF-Steuerelemente** die gängigen Steuerelemente, die Sie auch von Access her kennen – aber auch einige neue Kandidaten. Unter **Alle WPF-Steuerelemente** finden Sie die komplette Sammlung. Fügen Sie hier wie in Bild 3 eine Schaltfläche zum WPF-Fenster hinzu. Sie erkennen direkt, dass nicht nur die Schaltfläche im Fenster erscheint, sondern auch, dass der Designer ein Element zur XAML-Datei des Fensters hinzugefügt hat, und zwar in das **Grid**-Element:

```
<Button Content="Button" HorizontalAlignment="Left"
Margin="36,34,0,0" VerticalAlignment="Top" Width="75"/>
```

Dazu erhält das Steuerelement direkt einige Werte für Standardeigenschaften wie die horizontale und vertikale Ausrichtung (**HorizontalAlignment** und **VerticalAlignment**), die Ränder (**Margin**) und die Breite (**Width**). Die Beschriftung landet im Attribut **Content**. Klicken Sie nun auf **F5**, starten Sie die WPF-Anwendung und das Fenster

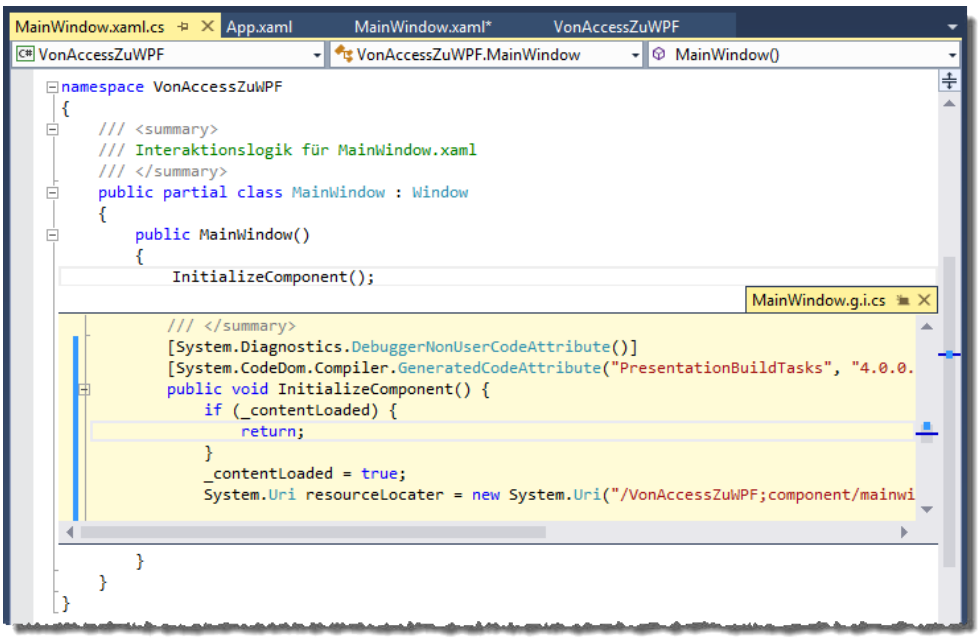


Bild 2: Der Code hinter einer Anweisung wird farbig hinterlegt eingeblendet.

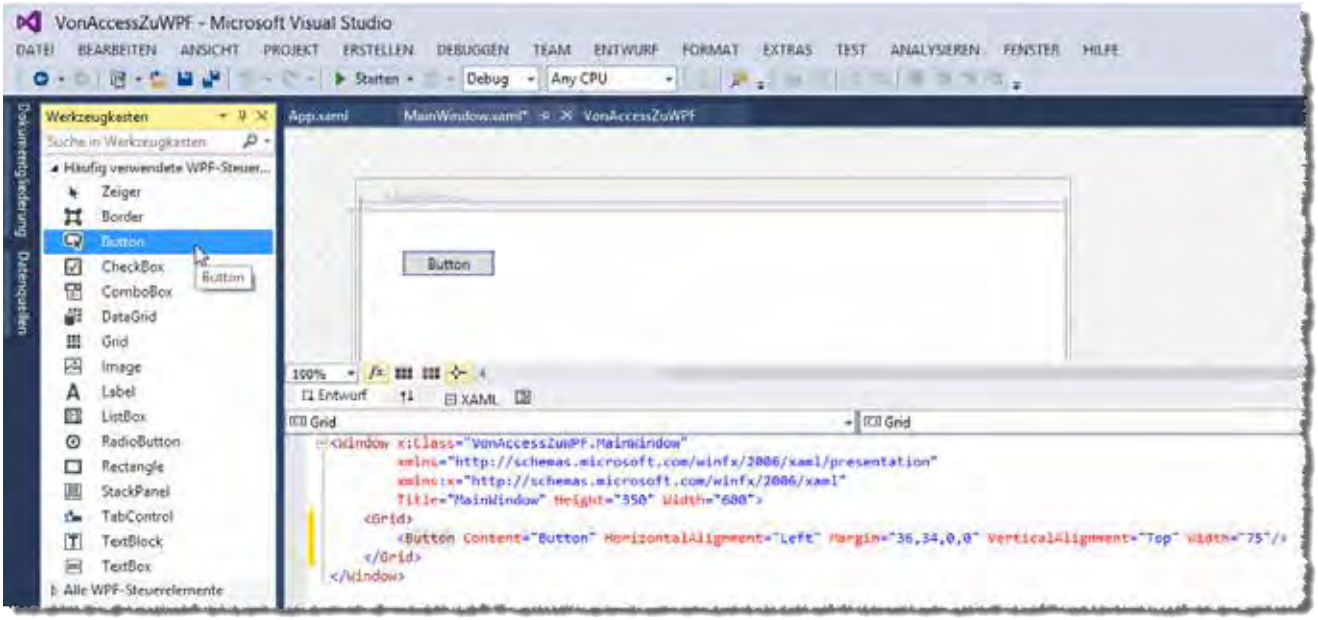


Bild 3: Neues Steuerelement im Entwurf und im XAML-Code

mit der Schaltfläche erscheint – diese ist allerdings noch ohne Funktion (siehe Bild 4).

Startfenster ändern

In einer Access-Anwendung ist standardmäßig kein Formular als Startformular der Anwendung voreingestellt. Das ist kein Problem: Sie können ja dort beispielsweise ein benutzerdefiniertes Ribbon anzeigen, mit dem der Benutzer die gewünschten Elemente der Benutzeroberfläche öffnen kann.

Alternativ wählen Sie für die Eigenschaft **Startformular** der Access-Optionen das beim Starten der Anwendung anzuzeigende Formular an. Bei einer WPF-Anwendung müssen Sie zwingend ein Startfenster angeben, denn sonst würde beim Start ja überhaupt keine Benutzeroberfläche erscheinen. Aus diesem Grund enthält ein neues WPF-Projekt ja auch gleich nach dem Start ein solches Fenster.

Wie aber gehen Sie vor, wenn Sie beispielsweise ein anderes Fenster beim Start der Anwendung anzeigen möchten? Dazu müssen Sie einfach nur den Inhalt der Datei **App.xaml** ändern. Dort gibt es im **Application**-Element ein Attribut namens **StartupUri**, das standardmäßig den Wert **MainWindow.xaml** enthält. Geben Sie hier einfach den Namen des beim Start anzuzeigenden Fensters ein (siehe Bild 5). Auch zu **App.xaml** gibt es eine entsprechende C#-Datei namens **App.xaml.cs**. Die **App.xaml**-Datei ist wichtig, weil sie der Startpunkt der Anwendung ist und, wie oben bereits beschrieben, den Namen des anzuzeigenden Fensters enthält.

Startformular vs. StartupUri

Im Prinzip entspricht das für das Attribut **StartupUri** angegebene Fenster dem Startformular, das Sie in den Optionen einer Access-Datenbank angeben können. In beiden Fällen wird das entsprechende Fenster beziehungsweise Formular aufgerufen.

Bild 4: Erster Test des neuen Fensters

Beispielprojekt: Den Code für die Beispiele der folgenden Abschnitte finden Sie im Verzeichnis **Access-ZuWPF_Startmethode**.

Und AutoExec?

Unter Access haben Sie auch die Möglichkeit gehabt, ein Makro namens **AutoExec** anzugeben, um direkt beim Start Aktionen auszuführen. Dieses muss nicht zwangsläufig ein Formular öffnen, sondern kann auch Makroaktionen ausführen oder eine VBA-Funktion aufrufen.

In WPF-Projekten haben Sie eine ähnliche Möglichkeit. Dazu bearbeiten Sie die Datei **App.xaml**, indem Sie im Element **Application** das Attribut **StartupUri** entfernen und das Element **Startup** hinzufügen. Das gelingt übrigens prima per Intelli-Sense (siehe Bild 6).

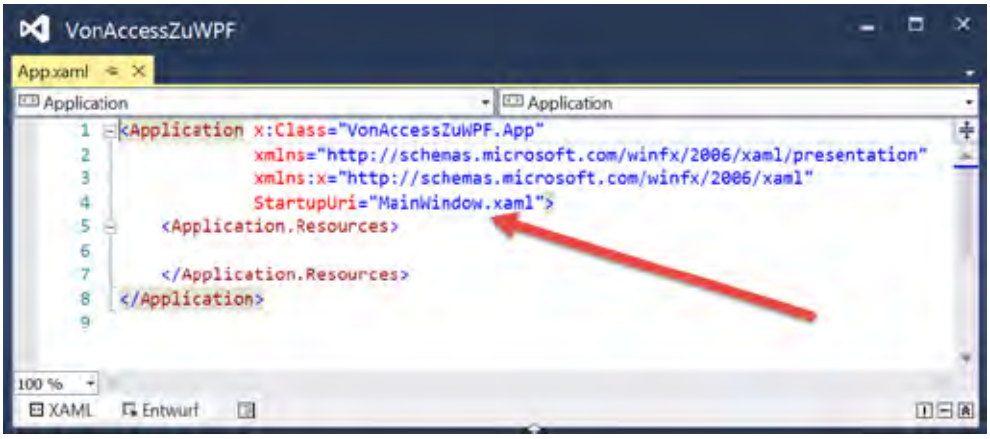


Bild 5: Angabe des Startfensters

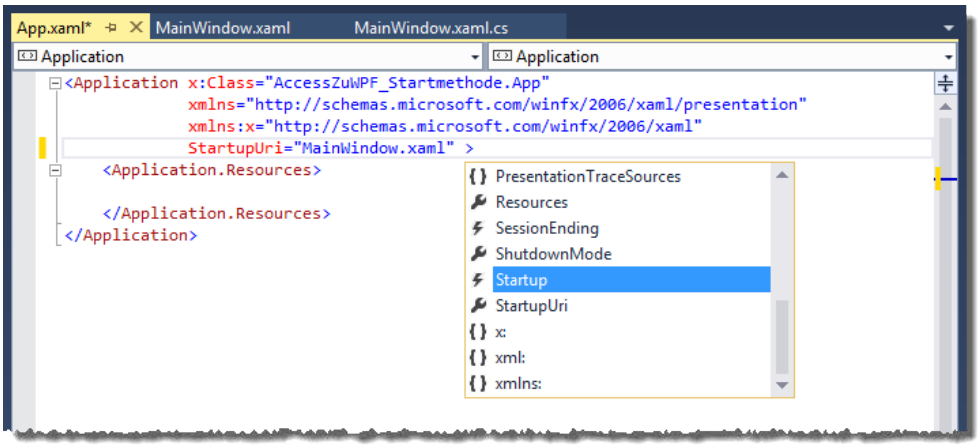


Bild 6: Auswahl des Attributs zur Angabe einer **Startup**-Methode

Nun müssen Sie nur noch die entsprechende Methode anlegen, und zwar in der Datei **App.xaml.cs**. Der Code dieser Klasse sieht dann wie folgt aus.

```
namespace AccessZuWPF_Startmethode {
    public partial class App : Application {
        private void Application_Startup(object sender,
                                         StartupEventArgs e)
        {
            MainWindow wnd = new MainWindow();
            wnd.Title = "Hauptfenster";
            wnd.Show();
        }
    }
}
```

Die hier enthaltene Methode **Application_Startup** erzeugt zunächst ein neues Objekt auf Basis unserer Fenster-Klasse **MainWindow** und weist dieses der Variablen **wnd** zu. Dann stellt sie den Titel auf Hauptfenster ein und blendet das Fenster schließlich mit der **Show**-Methode ein. Damit erhalten Sie auch gleich einen der Vorteile dieser Variante: Sie können die Eigenschaften des Fensters bereits vor dem Anzeigen ändern. Sie sehen also, dass Sie die Eigenschaften von Fenstern und Steuerelementen nicht nur über die Entwurfsansicht und den XAML-Code sowie über das Eigenschaften-Fenster anpassen können,

Datenzugriff mit ADO.NET, Teil 1

Wer lange mit Access gearbeitet hat, dem ist der Datenzugriff über die DAO-Bibliothek mit dem Recordset-Objekt, das Bearbeiten mit AddNew, Update und Edit sowie das Formulieren von SQL-Aktionsabfragen für die Execute-Anweisung in Fleisch und Blut übergegangen. Nun heißt es umdenken: Unter der Datenzugriffsbibliothek ADO.NET sieht vieles anders aus. Diese Artikelreihe zeigt, wie Sie die unter Access gelernten Abläufe auch mit Visual Studio unter C#/ADO.NET programmieren können.

ADO.NET ist nicht etwa der Nachfolger der zwischenzeitlich aufgekommenen ADODB-Bibliothek, die Sie auch in Access nutzen konnten (und können). Es ist eine komplett neue Bibliothek, die Microsoft für das .NET-Framework geschaffen hat.

ADO.NET bietet teilweise Datenzugriff mit offener Verbindung, teilweise verbindungslosen Zugriff an. Unter Access können Sie beispielsweise ein ADODB-Recordset mit den Daten einer Tabelle füllen und dann die Verbindung lösen, um die Daten zu verändern, ohne dass sich dies auf die zugrunde liegende Tabelle auswirkt. Dies geschieht mit vielen Objekten unter ADO.NET genauso.

Provider

Unter Access konnten Sie entweder direkt über die **CurrentDb**- oder die **OpenDatabase**-Funktion ein **Database**-Objekt referenzieren und dann mit Methoden wie **Execute**, **OpenRecordset** et cetera schreibend oder lesend/schreibend auf die Daten zugreifen. Oder Sie haben per ODBC mit einer entsprechenden Verbindungszeichenfolge auf weitere Datenbanken wie MySQL oder Microsoft SQL Server zugegriffen. Unter C# sieht dies etwas anders aus.

Hier verwenden Sie zwar auch noch eine Verbindungszeichenfolge, jedoch legen Sie sich bereits zuvor auf einen Datenprovider fest.

Es gibt beispielsweise die folgenden Provider:

- **SqlClient**: Microsoft SQL Server
- **OleDb**: Zugriff über einen OleDb-Provider, beispielsweise für den Zugriff auf Access-Datenbanken.
- **Odbc**: Zugriff auf Datenbanken über den jeweiligen ODBC-Treiber, zum Beispiel auf MySQL-Datenbanken oder ältere Access-Datenbanken wie Access 97.

Je nachdem, welchen Sie benötigen, fügen Sie Ihrer Klasse einen Verweis auf den jeweiligen Namespace hinzu:

```
//Für OleDb:
using System.Data.OleDb;
//Für Odbc:
using System.Data.Odbc;
//Für SQL Server:
using System.Data.SqlClient;
```

Beispieldatenbank

Für die Beispiele dieses Artikels verwenden wir eine Access-Datenbank, da wir diese einfach als Datei mitliefern können. Das heißt, dass wir den **OleDb**-Provider nutzen wollen:

```
using System.Data.OleDb;
```

Die Beispieldatenbank ist die Südsturm-Datenbank, die viele Leser vermutlich schon vom Magazin **Access im Unternehmen** kennen.

Die Datenbank enthält einige Tabellen, die wir für die folgenden Beispiele nutzen werden. Bild 1 zeigt das Datenmodell dieser Datenbank.

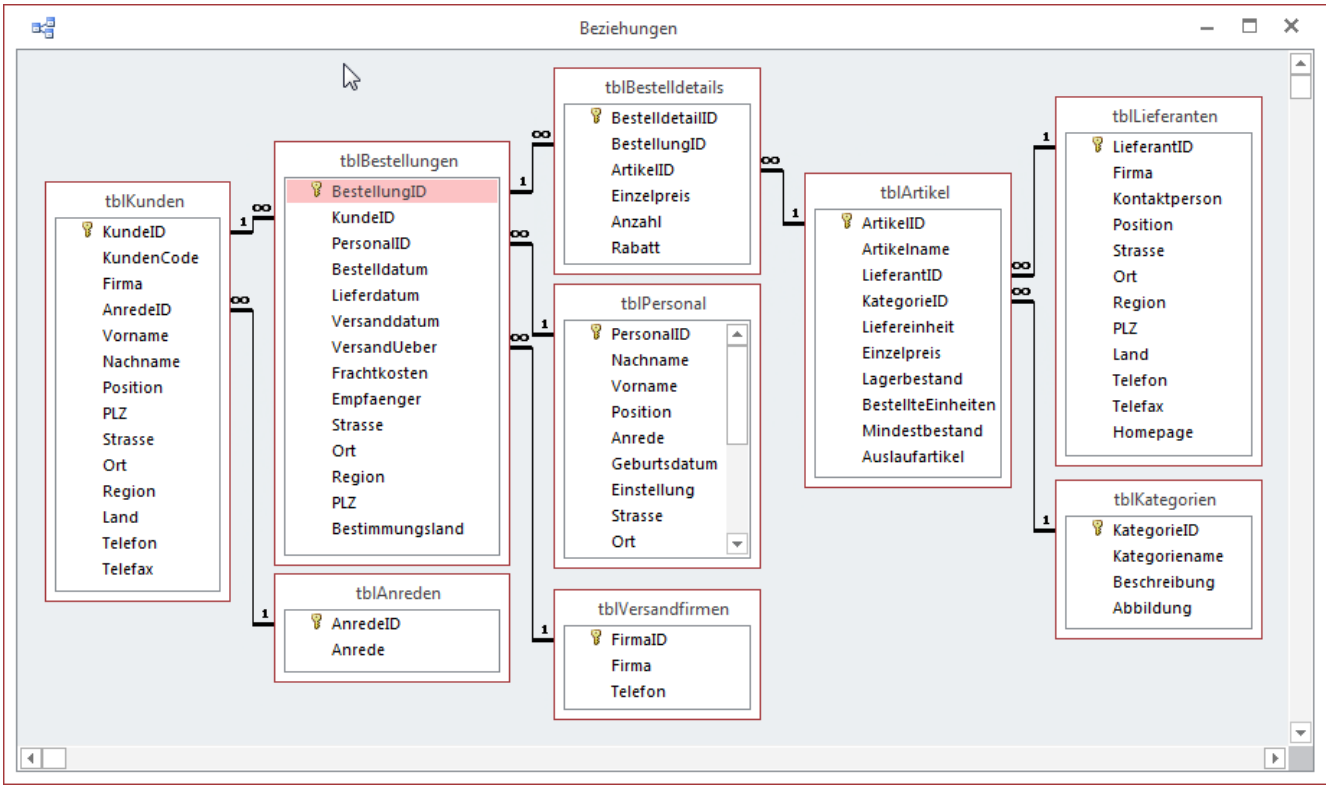


Bild 1: Datenmodell der Beispieldatenbank

Speicherort der Datenbank

Während der Tests werden wir die Anwendung von Visual Studio aus mit **F5** starten. Die **ADONET.exe**-Datei wird dann im Unterverzeichnis **\bin\debug** des Projektordners erstellt werden.

Damit wir ohne weitere Pfadangaben auf die Datei **Suedsturm.mdb** zugreifen können, fügen wir auch die Datenbankdatei zu diesem Verzeichnis hinzu.

Beispielprojekt

Als Beispielprojekt verwenden wir zunächst eine Konsolenanwendung (siehe Ordner **ADONET** im Download). Die einzelnen Beispiele können Sie jeweils in die Methode **Main** einfügen und ausführen.

Verbindung aufbauen

Grundlage für den Zugriff auf eine Datenbank ist der Aufbau einer Verbindung. Dies erledigen Sie unter C# wie in

```
static void Main(string[] args)
{
    string Connectionstring = "Provider=Microsoft.Jet.OLEDB.4.0;Data Source=Suedsturm.mdb";
    OleDbConnection cnn = new OleDbConnection(Connectionstring);
    cnn.Open();
    Console.WriteLine("Provider: {0}", cnn.Provider);
    Console.WriteLine("DataSource: {0}", cnn.DataSource);
    Console.ReadLine();
    cnn.Close();
}
```

Listing 1: Herstellen einer Verbindung und Ausgeben einiger Verbindungseigenschaften

der Methode aus Listing 1. Die Methode definiert zunächst eine Verbindungszeichenfolge, wie Sie sie möglicherweise auch von **ADODB** kennen.

Dann erstellt die Methode ein neues Objekt des Typs **OleDbConnection** mit der Verbindungszeichenfolge als Parameter und speichert den Verweis darauf in der Variablen **cnn**. Dessen **Open**-Methode öffnet schließlich die Verbindung und folgenden Anweisungen geben den **Provider** (**Microsoft.Jet.OLEDB.4.0**) und die **DataSource** (**Suedsturm.mdb**) in der Konsole aus.

Objekte nach Datenquelle

Es gibt unter C# kein **Connection**-Objekt wie unter Access, das Sie mit einer datenquellenspezifischen Verbindungszeichenfolge versehen. Stattdessen finden Sie hier verschiedene Objekte vor, zum Beispiel die folgenden:

- **OleDbConnection**
- **SqlConnection**
- **OdbcConnection**

Gleiches gilt auch für die übrigen Objekte weiter unten vorgestellten Objekte: Wenn Sie die Beispiele also nicht mit Access, sondern mit einer anderen Datenquelle ausprobieren möchten, ersetzen Sie das Präfix des jeweiligen Objekts (also etwa **OleDb...** durch **Sql...**, wenn Sie auf den SQL Server zugreifen möchten).

Aktionsabfragen ausführen

Unter VBA gibt es mehrere Möglichkeiten, Aktionsabfragen auszuführen. Dabei führen Sie die angegebene SQL-Anweisung (**INSERT INTO**, **UPDATE**, **DELETE** oder **SELECT INTO**) zum Beispiel mit der **Execute**-Methode des **Database**-Objekts oder mit der **RunSQL**-Methode des **DoCmd**-Objekts aus:

```
'VBA-Code:
Dim db As DAO.Database
Set db = CurrentDb
db.Execute "DELETE FROM tblArtikel", dbFailOnError
```

oder

```
'VBA-Code:
DoCmd.RunSQL "DELETE FROM tblArtikel"
```

Es gibt auch noch eine ADODB-Methode, die wir hier nicht mehr aufführen wollen. Unter C# verwenden wir das Objekt **OleDbCommand** (beziehungsweise **SqlCommand** oder **OdbcCommand** für andere Datenquellen als Access), um eine Aktionsabfrage abzusetzen. Ein Beispiel sieht wie

```
...
string Connectionstring = "Provider=Microsoft.Jet.OLEDB.4.0;Data Source=Suedsturm.mdb";
int AnzahlGeaenderteDatensaetze;
OleDbConnection cnn = new OleDbConnection(Connectionstring);
OleDbCommand cmd = new OleDbCommand();
cmd.CommandText = "UPDATE tblArtikel SET Artikelname = 'Chai (Tee)' WHERE Artikelname = 'Chai'";
cmd.Connection = cnn;
cnn.Open();
AnzahlGeaenderteDatensaetze = cmd.ExecuteNonQuery();
Console.WriteLine("Geänderte Datensätze: {0}", AnzahlGeaenderteDatensaetze);
Console.ReadLine();
cnn.Close();
...
```

Listing 2: Ausführen einer Aktualisierungsabfrage

in Listing 2 aus. Die dortigen Anweisungen (auszuprobieren innerhalb der **Main**-Methode des Konsolenprojekts) stellen wieder die Verbindungszeichenfolge zusammen. Die folgende Anweisung deklariert eine **int**-Variable namens **AnzahlGeaenderteDatensaetze** zum Erfassen der Anzahl der von der folgenden Änderungsabfrage betroffenen Datensätze.

Wie in der vorherigen Methode erstellen wir auch hier wieder ein **OleDbConnection**-Objekt. Außerdem benötigen wir ein Objekt des Typs **OleDbCommand** (referenziert mit der Variablen **cmd**).

Die Aktionsabfrage weisen wir der Eigenschaft **CommandText** des **OleDbCommand**-Objekts zu und legen über die Eigenschaft **Connection** das Verbindungsobjekt aus **cnn** fest. Die **Open**-Anweisung öffnet die Verbindung und die **ExecuteNonQuery**-Methode des **OleDbCommand**-Objekts führt die Aktionsabfrage aus. Sie liefert als Ergebnis die Anzahl der betroffenen Datensätze, die in der Variablen **AnzahlGeaenderteDatensaetze** landet. Die **Console.Write**-Methode gibt das Ergebnis in der Konsole aus.

DLookup und Co. unter ADO.NET

Wer sich in Access nicht mit SQL-Anweisungen auseinandersetzen wollte, hatte gute Karten: Abfragen ließen sich mit der Entwurfsansicht zusammenstellen und Detailinformationen zu einzelnen Datensätzen oder aggregierte Daten ließen sich mit den Domänenfunktionen wie **DLookup**, **DMax**, **DMin**, **DSum** et cetera zusammentragen.

Letztlich benötigten diese Funktionen aber fast genauso viele Informationen wie Sie zum Zusammenstellen einer entsprechenden SQL-Anweisung brauchten. Unter ADO.NET gibt es die hier genannten Vereinfachungen nicht mehr, dafür aber eine spezielle Methode des **OleDbCommand**-Objekts (beziehungsweise **OdbcCommand** oder **SqlCommand**), welche genau einen Wert aus einer Datensatzgruppe zurückliefert.

Diese Methode heißt **ExecuteScalar()**. Das Beispiel aus Listing 3 zeigt, wie es funktioniert. Die Anweisungen stellen wieder die Verbindungszeichenfolge zusammen und erstellen das **OleDbConnection**- und das **OleDbCommand**-Objekt. Die dem **OleDbCommand**-Objekt zugewiesene SQL-Abfrage braucht keine besonderen Voraussetzungen zu erfüllen, denn **ExecuteScalar** liefert den Wert des ersten Feldes des ersten gefundenen Datensatzes.

Wenn Sie also die Abfrage **SELECT ArtikelID, Artikelname FROM tblArtikel** aufrufen, erhalten Sie den Wert des Feldes **ArtikelID** für den ersten Datensatz – in diesem Fall **1**. Natürlich macht es keinen Sinn, der Abfrage mehr als ein Ausgabefeld mitzugeben, da ohnehin nur das erste Feld abgefragt wird.

```
...
string Connectionstring = "Provider=Microsoft.Jet.OLEDB.4.0;Data Source=Suedsturm.mdb";
OleDbConnection cnn = new OleDbConnection(Connectionstring);
OleDbCommand cmd = new OleDbCommand();
cmd.CommandText = "SELECT Count(*) FROM tblArtikel";
cmd.Connection = cnn;
cnn.Open();
int AnzahlArtikel;
AnzahlArtikel = (int)cmd.ExecuteScalar();
Console.WriteLine("Anzahl der Artikel: {0}", AnzahlArtikel);
Console.ReadLine();
cnn.Close();
```

Listing 3: Abrufen eines einzelnen Wertes über eine Abfrage

Objektorientierte Programmierung, Teil 1

Wer mit VBA programmiert, nutzt eigentlich schon eine Reihe der Elemente der objektorientierten Programmierung – zum Beispiel, wenn er Objekte auf Basis der Klasse Database oder Recordset erstellt und dann mit deren Methoden und Eigenschaften durch die Daten einer Tabelle navigiert. Auch benutzerdefinierte Klassen lassen sich mit VBA erstellen. Unter C# erhalten Sie jedoch Zugriff auf den kompletten Umfang der objektorientierten Programmierung. Dieser Artikel liefert einen Einstieg in die objektorientierte Programmierung mit C#.

Prozeduren starten ade

Eines müssen wir direkt klarstellen: Einfach mal eben wie im VBA-Editor eine Prozedur in einem Standardmodul programmieren und diese mit **F5** ausführen – das gelingt in Visual Studio unter C# nicht. Dort müssen Sie etwa in einer Konsolenanwendung zumindest die standardmäßig vorhandene Klasse verwenden, die dort enthaltene Prozedur **Main** mit den gewünschten Codezeilen füllen und auf **F5** drücken.

Die Klasse Program

Wenn Sie ein neues Projekt auf Basis der Vorlage **Konsolenanwendung** erstellen, erhalten Sie eine Datei mit einer Klasse namens **Program** mit einer Methode namens **Main**, die standardmäßig beim Start der Anwendung ausgeführt wird.

Von der Methode **Main** aus werden wir gleich starten, um unsere erste selbst erstellte Klasse zu instanzieren und zu nutzen.

Eine Klasse pro Datei

Jede Klasse sollte in einer eigenen Datei erstellt werden, um die Übersicht zu gewährleisten. Sollten Sie mehrere Klassen in einer Datei anlegen, finden Sie solche Klassen, deren Name sich von dem der Datei unterscheidet, nicht mehr so schnell über den Projektmappen-Explorer wieder.

Neue Klasse erstellen

Dementsprechend beginnen wir nun, indem wir eine neue Klasse erstellen. Dies erledigen wir über den Menüeintrag **Projekt|Klasse hinzufügen...** Dies öffnet den Dialog **Neues Element** hinzufügen und stellt die Vorlage direkt auf den Eintrag **Klasse** ein. Wir nennen die Klasse **Konto** und klicken dann auf Hinzufügen (siehe Bild 1). Alternativ öffnen Sie den Dialog **Neues Element** mit dem Kontextmenü-Eintrag **Hinzufügen|Klasse...** des Eintrags für das Projekt im Projektmappen-Explorer.

Die neue Klasse erscheint nun sowohl im Projektmappen-Explorer als auch mit dem vollständigen Quellcode im

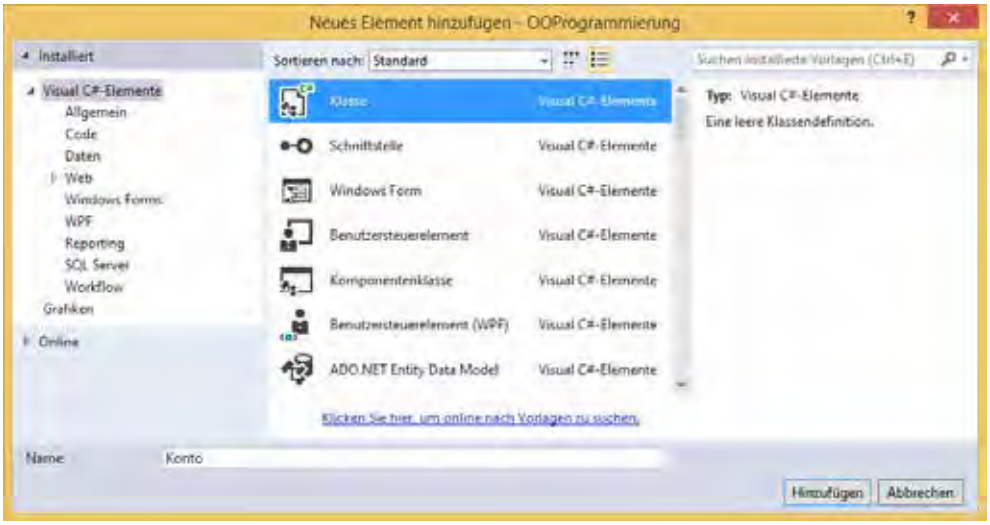


Bild 1: Hinzufügen einer neuen Klasse

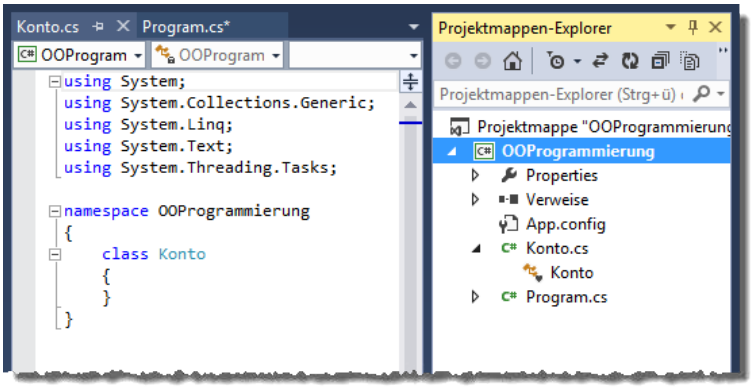


Bild 2: Die neue, noch unberührte Klasse **Konto**

Code-Fenster (siehe Bild 2). Abgesehen von den **using**-Anweisungen, dem **namespace**- und dem **class**-Element ist die Klasse noch jungfräulich:

```
namespace OOProgrammierung {
    class Konto {

    }
}
```

Benennung

Der Klassenname sollte möglichst genau beschreiben, welchem Objekt der Realität die Klasse entspricht – sofern dies gegeben ist. Soll die Klasse eine Person aufnehmen, heißt sie **Person**, soll sie wie in unserem Fall ein Konto aufnehmen, lautet der Klassenname **Konto**.

Wir verwenden kein Präfix, außerdem beginnt der Klassenname mit einem Großbuchstaben. Besteht der Klassenname aus mehreren zusammengesetzten Wörtern, sollten auch die folgenden Wörter jeweils wieder groß geschrieben werden.

Wenn Sie den Namen der Datei mit der Klasse später noch einmal ändern möchten, erledigen Sie da am besten über den Projektmappen-Explorer. Visual Studio ändert dann optional alle Verweise auf diese Klasse. Wenn Sie den Dateinamen **Konto** in **Bankkonto** ändern, benennt Visual Studio auch automatisch die Klasse im Code um.

Objekt instanzieren

Die Klasse ist zwar noch leer (sie hat noch keine Methoden, Eigenschaften et cetera), aber wir können diese dennoch bereits deklarieren und auch instanzieren. Dazu wechseln Sie nun zur Klasse **Program**. Dort erweitern Sie die Methode **Main** um zwei Zeilen:

```
static void Main(string[] args) {
    Konto konto;
    konto = new Konto();
}
```

Die erste Zeile deklariert ein neues Objekt namens **konto** mit dem Typ **Konto** (man beachte die Groß-/Kleinschreibung). Beides können Sie auch in einer einzigen Anweisung erledigen:

```
Konto konto = new Konto();
```

Die Deklaration reserviert den Speicher für das Objekt, die **new**-Anweisung erstellt das Objekt nach dem in der Klasse definierten Bauplan. Mit dem Objekt können wir nun noch nicht viel anfangen, da wir noch keine Methoden oder Eigenschaften hinzugefügt haben.

Sie können auch mehrere Objekte auf Basis der gleichen Klasse erstellen:

```
Konto konto1 = new Konto();
Konto konto2 = new Konto();
```

Sichtbarkeit der Klasse

Wenn Sie mit Access gearbeitet haben, kennen Sie die Klassenbibliotheken, die Sie üblicherweise per Verweis in ein VBA-Projekt eingebunden haben. Diese haben dann, sowohl über den Objektkatalog einsehbare als auch im Code referenzierbare Klassen bereitgestellt. Wenn Sie eine Klasse in einem C#-Projekt erstellen, um von anderen Anwendung darauf zuzugreifen, müssen Sie die Deklaration dieser Klasse mit dem Schlüsselwort **public** versehen:

```
public class Konto {

}
```

Wenn Sie das Schlüsselwort **public** nicht angeben, verwendet die Klasse die Standardeinstellung, welche dem Einsatz des Schlüsselworts **internal** entspricht und den Zugriff von außerhalb der Klassenbibliothek verwehrt.

Objekt freigeben

Objekte belegen Arbeitsspeicher. Unter VBA haben Sie eine Objektvariable durch Setzen auf den Wert **Nothing** freigegeben:

```
VBA-Code:
Set rst = Nothing
```

Unter C# stellen Sie das Objekt auf den Wert **null** ein:

```
konto = null;
```

Damit können Sie nun nicht mehr auf das Objekt zugreifen. Der Speicherplatz wird jedoch erst freigegeben, wenn die Garbage Collection startet und das Objekt löscht. Dies geschieht je nach Auslastung (bei hoher Auslastung seltener) und verfügbarem Speicherplatz (wenn der Platz knapp wird) oder bei Schließen der Anwendung.

Eigenschaften einer Klasse

Unsere Konto-Klasse soll verschiedene Eigenschaften bieten, zum Beispiel **Kontoinhaber**, **Bankleitzahl**, **Kontonummer**, **Kontostand** und **Dispositionsrahmen**. Um diese Eigenschaften nach dem Instanzieren eines Objekts auf Basis der Klasse **Konto** zu lesen und auch zu schreiben, deklarieren Sie diese wie folgt:

```
class Konto {
    public string Kontoinhaber;
    public string Kontonummer;
    public string Bankleitzahl;
    public decimal Kontostand;
    public decimal Dispositionsrahmen;
}
```

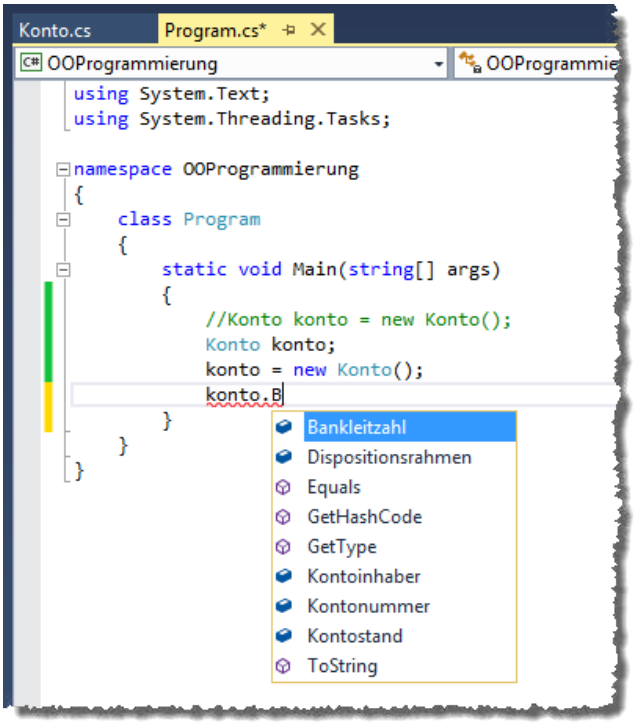


Bild 3: Eigenschaften per IntelliSense

Sie können dann von einer anderen Klasse aus, in diesem Fall **Program**, per IntelliSense auf die Eigenschaften zugreifen (siehe Bild 3). Dort sehen Sie noch einige weitere Methoden, beispielsweise **Equals** oder **GetType**.

Eigenschaften zuweisen und ausgeben

Das folgende Beispiel deklariert und instanziert das Objekt **konto** auf Basis der Klasse **Konto**, weist den Eigenschaften Werte zu und gibt diese dann in der Konsole aus:

```
Konto konto;
konto = new Konto();
konto.Kontoinhaber = "André Minhorst";
konto.Kontonummer = "1234567890";
konto.Bankleitzahl = "87654321";
konto.Kontostand = 100;
konto.Dispositionsrahmen = 1000;
Console.WriteLine("Kontoinhaber: {0}", konto.Kontoinhaber);
```

```
Console.WriteLine("Kontonummer:  {0}", konto.Kontonummer);
Console.WriteLine("Bankleitzahl: {0}", konto.Bankleitzahl);
Console.ReadLine();
```

Datenkapselung

Von VBA kennen Sie es vielleicht bereits: Die Eigenschaften einer Klasse werden nie als öffentliche Variablen deklariert, sondern mit den Schlüsselwort **Private**. Stattdessen erstellen Sie sogenannte **Property Get**- und **Property Let**- (für Werttypen) oder **Property Set**-Methoden (für Referenztypen), um auf die Inhalte der jeweiligen Variablen zuzugreifen. Hier haben wir dann zunächst eine private Variable für den Wert der Eigenschaft deklariert:

```
'VBA-Code:
Private m_Kontostand As Currency
```

Die **Property Let**-Methode erlaubte dann das Setzen des Wertes dieser Variablen von außerhalb der Klasse:

```
'VBA-Code:
Public Property Let Kontostand(curKontostand As Currency)
    m_Kontostand = curKontostand
End Property
```

Mit der **Get**-Methode konnte man den Wert hingegen lesen:

len, Methoden, Eigenschaften et cetera müssen innerhalb eines Namespaces eindeutig sein.

Also ändern wir beim Variablennamen den großen Anfangsbuchstaben in einen kleinen:

```
private string kontoinhaber;
```

Wenn Sie dies zu unübersichtlich finden, können Sie alternativ auch etwa den Namen mit Großbuchstaben verwenden und diesem einen Unterstrich voranstellen – also etwa **_Kontoinhaber**. Außerdem fügen wir eine öffentliche Eigenschaft mit einem **get**- und einem **set**-Block hinzu. Diese wird wieder mit großem Anfangsbuchstaben notiert:

```
public string Kontoinhaber {
    get {
        return kontoinhaber;
    }
    set {
        kontoinhaber = value;
    }
}
```

Sie enthält einen **get**-Block, der mit der **return**-Anweisung den aktuellen Inhalt der Variablen **kontoinhaber** zurückliefert. Der **set**-Block erwartet einen neuen Wert für die

Die Leseprobe dieses Artikels ist hier zu Ende.

Willst Du mehr?

ZUM SHOP

2.400 Seiten Know-how rund um die Datenbankentwicklung mit .NET

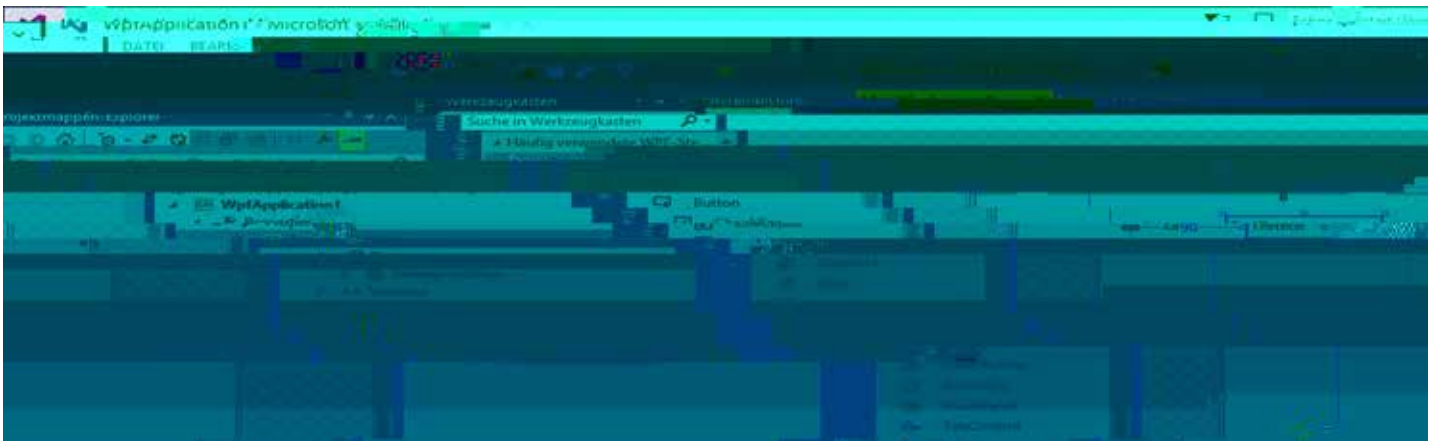
André Minhorst Verlag

Spare 20 EUR mit Code dbew20

AUSGABE 02/2015

DATENBANK ENTWICKLER

MAGAZIN FÜR DIE DATENBANKENTWICKLUNG MIT VISUAL STUDIO FÜR DESKTOP, WEB UND CO.



TOP-THEMEN:

C#-GRUNDLAGEN	Mit Zeichenketten arbeiten	SEITE 3
C#-GRUNDLAGEN	Auflistungen mit der ArrayList	SEITE 14
WPF	Steuerelemente anordnen	SEITE 24
WPF	Daten im DataGrid-Steuerelement anzeigen I	SEITE 36
PROGRAMMIEREN	Objektorientierte Programmierung: Grundlagen II	SEITE 42

André Minhorst Verlag

Mit Zeichenketten arbeiten

Schon VBA bietet eine Menge Funktionen für den Umgang mit Zeichenketten. Unter C# – wer hätte das gedacht – setzt Microsoft noch Einiges drauf. Einer der wichtigsten Unterschiede ist, dass eine Variable des Typs String nun nicht mehr einfach nur eine Zeichenkette aufnehmen kann, sondern sogar auf ein Objekt verweist, das den Text enthält und Methoden und Eigenschaften aufweist. Dieser Artikel liefert eine Einführung in den Umgang mit Zeichenketten unter C#.

Unter VBA ist eine Zeichenkette, also der Inhalt einer Variablen mit dem Datentyp **String**, ein Werttyp – genau wie die Zahlendatentypen oder Datum/Zeit. Unter C# (und den anderen .NET-Programmiersprachen) ist dies anders: Dort ist ein **String** ein Referenztyp, das heißt, dass die Objektvariable des Datentyps **String** lediglich die Speicheradresse enthält, hinter der sich der eigentliche Inhalt verbirgt. Außerdem bringt der **String** unter .NET die Eigenschaften eines Objekts mit, nämlich Methoden und Eigenschaften. Prinzipiell entspricht ein **String**-Objekt sogar einem Array, das Elemente des Datentyps **Char** enthält – der wiederum ein Werttyp ist.

Beispiele nachbauen

Wenn Sie die Beispiele dieses Artikels nachvollziehen möchten, legen Sie ein Projekt des Typs **Konsolenanwendung** mit der Programmiersprache **Visual C#** an. Die nachfolgenden Beispiele können Sie dann als Inhalt der Prozedur **Main** eintragen und mit **F5** ausführen.

Variable deklarieren und füllen

Für ein einfaches Beispiel, das einfach nur eine Variable des Typs **String** mit dem Namen **zeichenfolge** erstellt, mit einer Zeichenkette füllt und auf der Konsole ausgibt, sieht dies wie folgt aus:

```
namespace Zeichenketten {
    class Program     {
        static void Main(string[] args) {
            String zeichenfolge = "Beispieltext";
            Console.WriteLine(zeichenfolge);
            Console.ReadLine();
        }
    }
}
```

Die Anweisung zum Deklarieren und Füllen der Variablen können Sie, wie üblich, auch auf zwei Anweisungen aufteilen:

```
String zeichenfolge;
zeichenfolge = "Beispieltext";
```

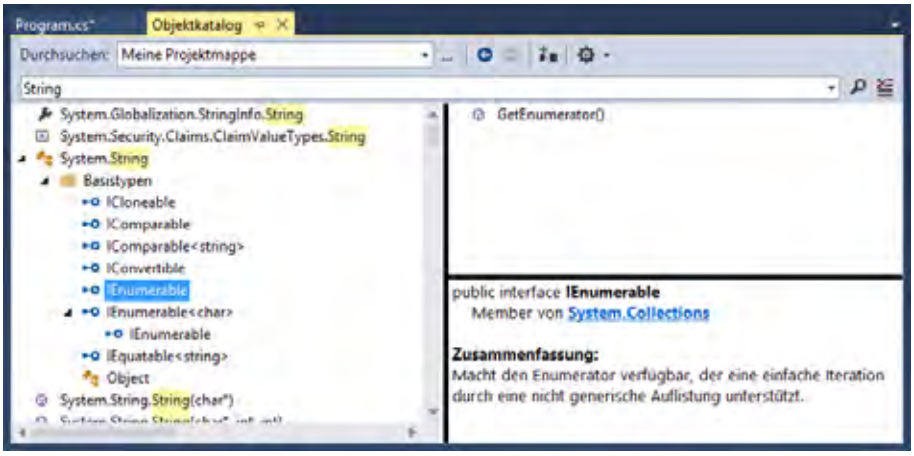


Bild 1: Ausgabe zweier **Write**-Anweisungen

Ein String ist ein Array?

Eingangs haben wir die Behauptung aufgestellt, dass ein **String**-Objekt Eigenschaften eines Array aufweisen würde. Dabei handelt es sich beispielsweise um die Möglichkeit, per Index auf die einzelnen Elemente des Typs **Char** zuzugreifen oder diese sogar per **foreach**-Schleife zu durchlaufen. Um diese Erkenntnis zu

gewinnen, gibt es zwei Möglichkeiten: Die erste ist es, sich einfach per IntelliSense die Eigenschaften und Methoden des **String**-Objekts anzeigen zu lassen. Oder Sie öffnen mit **Strg + Alt + J** den Objektkatalog und suchen dort nach der Zeichenkette **String**, was unter **System.String** schnell den richtigen Treffer liefert (siehe Bild 1).

Buchstaben per foreach-Schleife durchlaufen

Wenn Sie sich die Basistypen des **String**-Objekts ansehen, stoßen Sie schnell auf die **IEnumerable**-Schnittstelle. Ist diese vorhanden, können Sie die Elemente eines Objekts per **foreach**-Schleife durchlaufen.

Dies erledigen wir im folgenden Beispiel, wo wir eine **for-each**-Schleife über alle **Char**-Elemente des **String**-Objekts **zeichenfolge** durchlaufen und das jeweilige **Char**-Element mit der Variablen **buchstabe** referenzieren. Innerhalb der Schleife geben wir den Inhalt der **Char**-Variablen **buchstabe** zweimal aus – einmal unbehandelt und einmal zu einem Integer-Wert konvertiert:

```
String zeichenfolge = "Beispieltext";
foreach(Char buchstabe in zeichenfolge) {
    Console.WriteLine("{0} {1}", buchstabe,
        (int)buchstabe);
}
Console.ReadLine();
```

Das Ergebnis sieht wie in Bild 2 aus – neben dem Buchstaben erscheint der ASCII-Wert entsprechend der UTF-16-Kodierung für das jeweilige Zeichen.

Statt **int** hätten wir hier auch den Datentyp **byte** verwenden können:

```
(byte)buchstabe
```

Zeichenkette mit for durchlaufen

Das folgende Beispiel zeigt, wie Sie die Anzahl der enthaltenen Zeichen in einer **String**-Variablen ermitteln. Mit diesem Wert können wir dann eine **for**-Schleife definieren, die mit

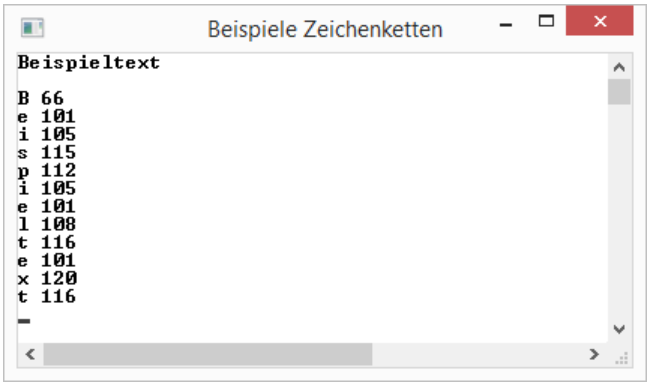


Bild 2: Ausgabe der **Char**-Elemente eines **String**-Objekts als Text und Integer

jedem Durchlauf das aktuelle **Char**-Element über den Index der **String**-Variablen **zeichenfolge** ermittelt und der Variablen **buchstabe** zuweist.

Auch hier landen der Buchstabe und der ASCII-Code in der Ausgabe:

```
String zeichenfolge = "Beispieltext";
int anzahl = zeichenfolge.Length;
for (int i = 0; i < anzahl; i++) {
    Char buchstabe = zeichenfolge[i];
    Console.WriteLine("{0} {1}", buchstabe,
        (int)buchstabe);
}
Console.ReadLine();
```

Dies ist schon einmal erheblich einfacher als unter VBA – dort hätten Sie erstens nur eine **For...Next**-Schleife verwenden können, um auf die einzelnen Zeichen zuzugreifen, und zweitens hätten wir für jeden einzelnen Zugriff die **Mid**-Funktion auf die Zeichenkette anwenden müssen:

```
For i = 0 To Len(strZeichenkette)
    Debug.Print Mid(strZeichenkette, i, 1)
Next i
```

Zur Ermittlung des ASCII-Codes hätten wir dort noch die Funktion **Asc** verwenden müssen, statt einfach nur den Buchstaben in den Zahlentyp **int** oder **byte** zu konvertieren.

Übrigens können Sie auch einfach über den Index auf ein Zeichen eines String-Objekts zugreifen – beispielsweise wie folgt auf das erste Zeichen:

```
string Vorname = "André";
char Buchstabe = Vorname[0];
```

Unterschied zwischen String und Char

Es gibt eine Reihe Unterschiede zwischen **string**- und **char**-Objekten. Einen haben Sie schon kennen gelernt: Ein Char ist ein Element eines Arrays, das in einer **string**-Variablen gespeichert wird. Ein anderer ist, dass nicht beide etwa beim Zuweisen in Anführungszeichen eingefasst werden.

Beim **string** verwenden Sie das übliche Anführungszeichen:

```
string zeichenfolge;
zeichenfolge = "Beispieltext";
```

Bei einer **Char**-Variable, die ja nur ein Zeichen aufnehmen kann, verwenden Sie hingegen einfache Anführungszeichen:

```
Char zeichen;
zeichen = 'A';
```

Dies nur, damit Sie sich nicht über Fehlermeldungen wundern, wenn Sie einmal einer **Char**-Variablen ein Zeichen in Anführungszeichen übergeben möchten (**Eine implizite Konvertierung von Typ 'string' in 'char' ist nicht möglich.**).

Zeichenfolge aus Char-Elementen zusammensetzen

Eine Zeichenfolge müssen Sie nicht immer direkt zuweisen; Sie können diese auch aus einem Array von **char**-Objekten zusammenstellen. Dazu brauchen Sie einfach nur das **char**-Array als Parameter beim Erstellen einer neuen **String**-Instanz anzugeben:

```
char[] chars = { 'A', 'n', 'd', 'r', 'é' };
String strChars = new String(chars);
Console.WriteLine("String aus Char-Array: {0}", strChars);
//Ausgabe:
//String aus Char-Array: André
```

Position einer Teilzeichenkette ermitteln

Unter VBA war hierzu die Funktion **InStr** angezeigt, wenn Sie das erste Auftreten einer Teilzeichenfolge ermitteln wollten, und **InStrRev**, wenn das erste Auftreten der Teilzeichenfolge von hinten gefragt war.

Das erste Auftreten einer Zeichenfolge in einer **String**-Variablen ermitteln Sie mit der Methode **IndexOf**, die als Parameter die zu suchende Teilzeichenkette erwartet:

```
String zeichenfolge;
zeichenfolge = "Beispieltext";
int position = zeichenfolge.IndexOf("spiel");
Console.WriteLine("Position von 'spiel': {0}", position);
//Ausgabe:
//3
```

Die Zeichenfolge **spiel** beginnt im Beispieltext mit dem vierten Zeichen, was dem Indexwert **3** (nullbasiert) entspricht.

Die VBA-Funktion **InStrRev** bilden Sie unter C# mit der Methode **LastIndexOf** ab. Diese sucht das erste Auftreten einer Zeichenfolge von hinten:

```
zeichenfolge = "blablabla";
position = zeichenfolge.LastIndexOf("bla");
Console.WriteLine("Position: {0}", position);
```

Zeichenfolge nicht gefunden

Findet **IndexOf** keine passende Teilzeichenkette, liefert es den Wert **-1** zurück:

```
zeichenfolge = "blablabla";
position = zeichenfolge.IndexOf("bla");
while (position != -1) {
    Console.WriteLine("Position: {0}", position);
    position = zeichenfolge.IndexOf("bla", position + 1);
}
```

```
//Ausgabe:
//Position: -1
```

Alle Auftreten einer Zeichenfolge finden

Unter VBA haben Sie hier eine **Do While**-Schleife genutzt, die so lange lief, bis der Wert für die gefundene Position den Wert **0** angenommen hat.

Hierzu ist anzumerken, dass **InStr** 1-basiert ist:

```
Dim intPosition As Integer
intPosition = InStr(1, "blablabla", "bla")
Do While Not intPosition = 0
    Debug.Print intPosition
    intPosition = InStr(intPosition + 1, "blablabla", "bla")
Loop
```

Unter C# hilft eine **while**-Schleife weiter sowie eine Überladung der Methode **IndexOf** – nämlich diejenige, die als zweiten Parameter die Startposition für die Suche erwartet:

```
zeichenfolge = "blablabla";
position = zeichenfolge.IndexOf("bla");
while (position != -1) {
    Console.WriteLine("Position: {0}", position);
    position = zeichenfolge.IndexOf("bla", position + 1);
}
//Ausgabe:
//0
//3
//6
```

Hier ist zu beachten, dass **IndexOf** – wie oben beschrieben – den Wert **-1** zurückliefert, wenn kein weiteres Auftreten mehr gefunden wird. Die Schleife verwendet dementsprechend den Ausdruck **position != -1** als Kriterium für die Fortsetzung.

Zeichenfolgen vergleichen

Zum Vergleichen zweier Zeichenfolgen liefert C# gleich mehrere Möglichkeiten, die wir uns nun ansehen.

Die erste Variante ist die Methode **Equals** des **string**-Objekts. Diese verwenden Sie als Methode des ersten zu vergleichenden **string**-Objekts und geben das zweite **string**-Objekt als Parameter an:

```
string str1 = "André";
string str2 = "André";
Console.WriteLine("{0} Equals {1}: {2}", str1.ToString(),
    str2.ToString(), str1.Equals(str2));
//Ausgabe:
//André Equals André: True
```

Diese Methode vergleicht tatsächlich die in den beiden Variablen gespeicherten Objekte. Aber wie können diese gleich sein – wir haben doch unabhängig voneinander zwei **string**-Objekte angelegt und gefüllt? Der Grund ist: Die Strings werden intern gespeichert, aber nur jeweils einmal. Haben Sie also bereits eine **string**-Variable mit dem Wert **André** gefüllt und legen eine weitere Variable mit diesem Wert an, dann verweisen beide **string**-Variablen auf das gleiche **string**-Objekt.

Anders funktioniert die Vergleichsmethode **Compare**. Diese ist eine Methode der **String**-Klasse und erwartet die beiden zu vergleichenden **string**-Variablen als Parameter. Sie wird beispielsweise so aufgerufen:

```
Console.WriteLine(String.Compare(str1, str2);
```

Sind beide Zeichenketten gleich, liefert **Compare** nicht den Wert **True**, sondern den Wert **0**. Andere Ergebnisse sind **-1**, wenn der erste Ausdruck kleiner als der zweite ist (zum Beispiel **a** und **b**) und **1**, wenn der erste Ausdruck kleiner als der zweite ist (**b** und **a**). Dabei bedeutet kleiner, dass die verglichenen Buchstaben sich im Alphabet weiter vorn befinden. Bezüglich kleiner und großer Buchstaben gilt, dass kleinere Buchstaben kleiner sind. **a** ist also kleiner als **A** und **A** ist kleiner als **b**.

Die Methode **CompareOrdinal** funktioniert ähnlich wie **Compare** und liefert bei Gleichheit den Wert **0** zurück. Wenn

jedoch beim zeichenweisen Durchlaufen der erste Unterschied festgestellt wird, ermittelt die Methode die Differenz bezüglich des ASCII-Codes der Zeichen.

Im folgenden Beispiel stößt die Methode also beim dritten Zeichen auf einen Unterschied. Sie ermittelt die ASCII-Codes der beiden Zeichen (**67** für **C** und **99** für **c**) und gibt als Ergebnis die Differenz aus, also **-32**:

```
str1 = "ABC";
str2 = "ABc";
Console.WriteLine("{0} CompareOrdinal {1}: {2}",
    str1.ToString(), str2.ToString(),
    String.CompareOrdinal(str1, str2));
//Ausgabe:
//ABC CompareOrdinal ABc: -32
```

Die Überladung dieser Methode konzentriert sich auf Teilzeichenketten der zu vergleichenden **string**-Objekte:

```
String.CompareOrdinal(str1, 3, str2, 3, 1));
```

Der erste und dritte Parameter nehmen wieder die zu untersuchenden Zeichenfolgen entgegen. Der zweite und vierte Parameter legen fest, ab welcher Position diese untersucht werden sollen, der fünfte, wie lang die zu vergleichenden Zeichenketten sein sollen.

Mid, Left, Right mit Substring

Diese drei VBA-Funktionen erlauben es, eine beliebige Teilzeichenfolge, eine Teilzeichenfolge beginnend mit dem ersten oder eine Teilzeichenfolge endend mit dem letzten Zeichen einer Zeichenkette zu ermitteln.

Unter C# ist das etwas komplizierter:

```
String zeichenfolge = "1234567890";
//erste vier Zeichen
Console.WriteLine(zeichenfolge.Substring(0, 4));
//letzte vier Zeichen
Console.WriteLine(zeichenfolge.Substring(6, 4));
//mittlere vier Zeichen
Console.WriteLine(zeichenfolge.Substring(3, 4));
```

Die Funktionen **Left**, **Right** und **Mid** gibt es also leider nicht. Sie könnten diese zwar nachbauen, aber wir sollten uns doch auf die vorhandenen Möglichkeiten konzentrieren – zum Beispiel die folgende.

Womit beginnt oder endet die Zeichenfolge?

Und wer **Left** oder **Right** nur dazu verwenden will, einen Teil der Zeichenkette von vorn oder hinten zu extrahieren, um diesen mit einer anderen Zeichenkette zu vergleichen, der wird mit den folgenden beiden Methoden ohnehin viel leichter glücklich:

- **StartsWith** ist eine Methode eines **string**-Objekts, das die ersten Zeichen mit der als Parameter übergebenen Zeichenkette vergleicht.

- **EndsWith** erledigt dies für das Ende der Zeichenkette. Beide berücksichtigen die Groß- und Kleinschreibung.

Im ersten Beispiel prüfen wir, ob die Zeichenkette aus **str1** mit der Zeichenkette **str2** beginnt:

```
string str1 = "André Minhorst";
string str2 = "And";
Console.WriteLine("'{0}' beginnt mit '{1}': {2}",
    str1, str2, str1.StartsWith(str2));
//Ausgabe:
//'André Minhorst' beginnt mit 'And': True
```

Das zweite Beispiel vergleicht das Zeichenkettenende mit dem Inhalt von **str3**:

```
string str1 = "André Minhorst";
string str3 = "rst";
Console.WriteLine("'{0}' endet mit '{1}': {2}",
    str1, str3, str1.EndsWith(str3));
//Ausgabe:
//'André Minhorst' endet mit 'rst': True
```

Auflistungen mit der ArrayList

Von VBA kennen Sie als Auflistungsklassen zunächst die eingebaute Collection. Wer noch etwas weiter geht, bindet die Scripting Runtime Library ein und nutzt die Dictionary-Klasse zum Speichern von Auflistungen. Über den reinen Programmcode hinweg ließen sich diese Elemente unter VBA kaum nutzen – als Datenherkunft etwa für Formular oder Kombinationsfelder taugten sie nicht. Unter C# sieht dies anders aus. Es gibt eine ganze Reihe verschiedener Auflistungsklassen, die sich teilweise auch als Datenherkunft für Steuerelemente eignen. In diesem Artikel schauen wir uns die ArrayList als Vertreter der Auflistungsklassen an.

Die Auflistungsklasse **ArrayList** steckt im Namespace **System.Collections**. Diese fügen Sie mit der **using**-Anweisung zur Klasse hinzu – also so:

```
using System.Collections;
```

Auf diese Weise können Sie gleich auf das ArrayList-Objekt und seine Methoden und Eigenschaften zugreifen und müssen nicht immer den Namespace voranstellen. Ohne Angabe des Namespace mit der using-Anweisung wäre für die Deklaration beispielsweise diese Zeile nötig:

```
System.Collections.ArrayList arrayList;
```

Wenn Sie den Namespace zuvor wie oben beschrieben angegeben haben, reicht dies aus:

```
ArrayList arrayList;
```

Schnittstellen von Auflistungsklassen

Neben den eigentlichen **Collection**-Klassen enthalten die beiden Namespaces noch einige weitere Objekte. Einige davon sind Interfaces, also Schnittstellenklassen. Diese beschreiben die grundlegenden Methoden und Eigenschaften der **Collection**-Objekte. Die **Collection**-Objekte implementieren diese Schnittstellen, was kurz gefasst bedeutet, dass sie die für die Schnittstelle vorgegebenen Methoden, Eigenschaften und Ereignisse enthalten.

Diese Schnittstellenklassen sehen Sie sich am besten im Objektkatalog an (**Strg + Alt + J**). Wenn Sie dort unter **mscorlib** den Eintrag **System.Collections** auswählen, erhalten Sie die Übersicht aus Bild 1. Dort sehen Sie, dass zum Beispiel die Methode **Add** zum Hinzufügen eines Auflistungsobjekts aus der Schnittstelle **IList** stammt – genauso wie einige

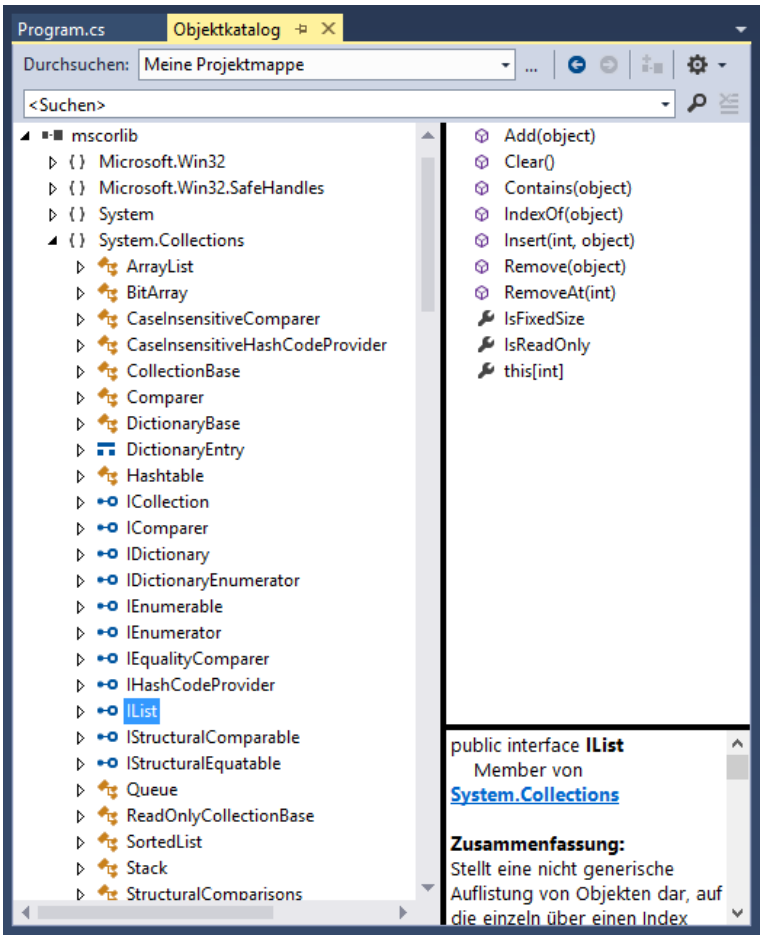


Bild 1: Schnittstelle einer Auflistungsklasse im Objektkatalog

andere Methoden. Die folgenden Schnittstellenklassen liefern die meisten Methoden und Eigenschaften, aus denen sich die verschiedenen Auflistungsklassen zusammensetzen:

- **ICollection**: Liefert neben anderen die **Count**-Eigenschaft zum Ermitteln der Anzahl der Elemente und die **CopyTo**-Methode, mit der Sie den Inhalt der Auflistungsklasse in ein Array kopieren können.

- **IList**: Liefert Methoden, um Listenelemente hinzuzufügen, zu entfernen oder deren Informationen zu ermitteln. Auflistungsklassen, die diese Schnittstelle implementieren, stellen ihre Elemente über einen Index bereit. Dies entspricht also etwa dem **Collection**-Objekt, das Sie von VBA kennen. Die Elemente der **IList**-Schnittstelle haben Sie bereits im Artikel **Von VBA Zu C#: Arrays** (**www.datenbankentwickler.de/6**) kennengelernt.

- **IDictionary**: Liefert ähnliche Methoden wie **IList**, aber ist für die Verwaltung von Name-Wert-Paaren ausgelegt. Dies entspricht dem **Dictionary**-Objekt der **Scripting Runtime Library**, das Sie möglicherweise unter VBA genutzt haben.

- **IEnumerable**: Steuert die Möglichkeit bei, die Elemente einer Auflistung per **foreach**-Schleife zu durchlaufen.

Diese Schnittstellenklassen liefern allerdings nicht alle Methoden und Eigenschaften etwa der **ArrayList**-Klasse – diese trägt selbst auch noch eigene Elemente bei.

ArrayList deklarieren und instanzieren

Ein **ArrayList**-Objekt können Sie in zwei Schritten deklarieren und instanzieren:

```
ArrayList arrayList;
arrayList = new ArrayList();
```

Wenn diese beiden Anweisungen ohnehin direkt nacheinander ausgeführt werden, können Sie dies auch in einer Zeile durchführen:

```
ArrayList arrayList = new ArrayList();
```

Elemente hinzufügen

Die einfachste Methode, ein Element zu einer **ArrayList** hinzuzufügen, ist die **Add**-Methode. Sie erwartet lediglich den hinzuzufügenden Wert beziehungsweise die Variable mit dem Objektverweis als Parameter. Die folgenden Anweisungen füllen das Array beispielsweise mit drei Zeichenketten:

```
arrayList.Add("Element 1");
arrayList.Add("Element 2");
arrayList.Add("Element 3");
```

Die **Add**-Methode fügt die Elemente immer hinten an die Liste an.

Index beim Hinzufügen ermitteln

Wenn Sie gleich beim Hinzufügen den Indexwert für das neue Element ermitteln wollen, speichern Sie das Ergebnis der **Add**-Methode in einer Integer-Variablen. Diesen können Sie dann wie folgt auf der Konsole ausgeben:

```
int indexNeuesElement = arrayList.Add("Element 4");
Console.WriteLine("Der Index des neuen Elements lautet:
{0}", indexNeuesElement);
```

Elemente direkt beim Initialisieren hinzufügen

Sie können auch direkt beim Initialisieren eine Liste der Elemente in geschweiften Klammern angeben:

```
arrayList = new ArrayList { "Element 1", "Element 2",
"Element 3" };
```

Anzahl der Elemente beim Erstellen angeben

Sie können beim Erstellen eines **ArrayList**-Objekts gleich die Anzahl der Einträge angeben – und zwar in Klammern hinter dem **ArrayList**-Schlüsselwort nach der **new**-Anweisung. Folgendes legt eine **ArrayList** mit drei Elementen an, aber nur die ersten beiden werden gefüllt. Bei der Ausgabe berücksichtigt die **foreach**-Schleife nur solche Elemente, die auch gefüllt wurden:

```
arrayList = new ArrayList(3);
arrayList.Add("Element 1");
arrayList.Add("Element 2");
foreach (string Element in arrayList) {
    Console.WriteLine("Element: {0}", Element);
}
```

Mit der Eigenschaft **Capacity** geben Sie die Anzahl der reservierten Elemente zurück oder legen diese fest – zum Beispiel so:

```
Console.WriteLine("Anzahl Elemente: {0}",
    arrayList.Capacity);
```

Element an bestimmter Stelle einfügen

Da die **Add**-Methode neue Elemente immer nur hinten an die Liste anhängt, benötigen wir noch eine Methode, die Elemente an einer bestimmten Position einfügt. Dies gelingt mit der Methode **Insert**. Diese erwartet als ersten Parameter den Index-Wert der Position, an der das neue Element eingefügt werden soll, als zweiten Parameter das einzufügende Element. Die folgenden Anweisungen legen zunächst zwei Elemente an und fügen dann ein weiteres Element an die Position des zweiten, sodass dieses an die dritte Stelle rutscht:

```
arrayList.Add("Element 1");
arrayList.Add("Element 2");
arrayList.Insert(1, "Element zwischen 1 und 2");
```

Da der Index der ArrayList 0-basiert ist, geben Sie als Position für den zweiten Eintrag den Wert **1** an.

Element entfernen

Wenn Sie gezielt ein Element aus einem Array entfernen möchten, benötigen Sie entweder dessen Inhalt oder dessen Index-Wert. Wenn Sie das Element kennen, können Sie es direkt über seinen Inhalt mit der **Remove**-Methode löschen:

```
arrayList.Add("Element 1");
arrayList.Add("Element 2");
arrayList.Add("Element 3");
arrayList.Remove("Element 2");
foreach (string eintrag in arrayList)
{
    Console.WriteLine(eintrag);
}
//Ausgabe:
//Element 1
//Element 2
```

Oder Sie kennen den Index des zu löschenden Elements. Diesen können Sie dann der **RemoveAt**-Methode als Parameter übergeben. Das folgende Beispiel entfernt beispielsweise das zweite von drei Elementen aus einer **ArrayList**:

```
arrayList.Add("Element 1");
arrayList.Add("Element 2");
arrayList.Add("Element 3");
arrayList.RemoveAt(1);
foreach (string eintrag in arrayList) {
    Console.WriteLine(eintrag);
}
//Ausgabe:
//Element 1
//Element 2
```

Mehrere Elemente entfernen

Wenn Sie mehrere zusammenhängende Elemente entfernen möchten, nutzen Sie die **RemoveRange**-Methode. Diese erwartet zwei Parameter: den Index des ersten und die Anzahl der zu entfernenden Elemente. Im folgenden Beispiel entfernt die Methode das zweite und dritte von vier Elementen:

```
arrayList = new ArrayList();
arrayList.Add("Element 1");
arrayList.Add("Element 2");
arrayList.Add("Element 3");
arrayList.Add("Element 4");
arrayList.RemoveRange(1, 2);
foreach (string eintrag in arrayList) {
    Console.WriteLine(eintrag);
}
```

```
//Ausgabe:
//Element 1
//Element 4
```

Elemente aus einer anderen ArrayList anfügen

Wenn Sie zwei **ArrayList**-Objekte verwenden und die Elemente des zweiten hinten an das erste Array anfügen möchten, verwenden Sie die Methode **AddRange**. Im folgenden Beispiel werden zwei **ArrayList**-Objekte mit je zwei Elementen gefüllt. Dann fügt die **AddRange**-Methode die Elemente von **arrayList2** hinten an **arrayList** an:

```
arrayList = new ArrayList();
arrayList.Add("Element 1");
arrayList.Add("Element 2");
arrayList2 = new ArrayList();
arrayList2.Add("Element 3");
arrayList2.Add("Element 4");
arrayList.AddRange(arrayList2);
```

Würden Sie nun alle Elemente von **arrayList** auf der Konsole ausgeben, erhielten Sie dieses Ergebnis:

```
Element 1
Element 2
Element 3
Element 4
```

Elemente aus einer anderen ArrayList einfügen

Auf ähnliche Weise fügen Sie die Elemente des zweiten **ArrayList**-Objekts an beliebiger Stelle in der ersten ArrayList ein. Dazu nutzen Sie diesmal jedoch die Methode **InsertRange**. Diese erwartet neben der Angabe der einzufügenden Liste noch den Index, an dessen Stelle die neuen Elemente eingefügt werden sollen:

```
arrayList = new ArrayList();
arrayList.Add("Element 1");
arrayList.Add("Element 2");
arrayList2 = new ArrayList();
arrayList2.Add("Element 3");
arrayList2.Add("Element 4");
arrayList.InsertRange(1,arrayList2);
foreach (String eintrag in arrayList) {
    Console.WriteLine(eintrag);
}
//Ausgabe:
//Element 1
//Element 3
//Element 4
//Element 2
```

Neue ArrayList aus Elementen einer ArrayList erstellen

Sie können auch aus einem oder mehreren Elementen einer ArrayList eine neue ArrayList erstellen. Dies geschieht mit der Methode **GetRange**. **GetRange** erwartet zwei Parameter: den Index des ersten zu exportierenden Elements und die Anzahl der Elemente. Das Ergebnis weisen Sie einfach einem anderen **ArrayList**-Objekt zu, wie hier **arrayList** zu **arrayList2**:

```
arrayList = new ArrayList();
arrayList.Add("Element 1");
arrayList.Add("Element 2");
arrayList.Add("Element 3");
arrayList2 = new ArrayList();
arrayList2 = arrayList.GetRange(1, 2);
foreach (string eintrag in arrayList2) {
    Console.WriteLine(eintrag);
}
//Ausgabe:
//Element 2
//Element 3
```

ArrayList mit Elementen einer beliebigen Collection füllen

Wenn Sie alle Elemente einer Collection an einer bestimmten Stelle eines **ArrayList**-Objekts einfügen und somit die dort vorliegenden Elemente überschreiben wollen, erledigen Sie dies mit der **SetRange**-Methode. Diese erwartet den Startindex für die zu ersetzenden Elemente der Ziel-**ArrayList**

sowie einen Verweis auf die Collection, welche die zu kopierenden Elemente liefert:

```
ArrayList zielArrayList = new ArrayList();
zielArrayList.Add("Element 1");
zielArrayList.Add("Element 2");
zielArrayList.Add("Element 3");
zielArrayList.Add("Element 4");
zielArrayList.Add("Element 5");
ArrayList quellArrayList = new ArrayList();
quellArrayList.Add("Element 6");
quellArrayList.Add("Element 7");
zielArrayList.SetRange(2, quellArrayList);
foreach (string Element in zielArrayList)
{
    Console.WriteLine("Element: {0}", Element);
}
//Ausgabe:
//Element 1
//Element 2
//Element 6
//Element 7
//Element 5
```

ArrayList in Array überführen

Sie können die Elemente der ArrayList auch in ein Array des Typs **Object** überführen. Dazu verwenden Sie die Methode **ToArray()** und schreiben das Ergebnis mit einer Variable des Typs **Object**:

```
arrayList = new ArrayList();
arrayList.Add("Element 1");
arrayList.Add("Element 2");
arrayList.Add("Element 3");
ArrayList arrayList6 = new ArrayList();
object[] array;
array = arrayList.ToArray();
foreach (string Element in array)
{
    Console.WriteLine("Element: {0}", Element);
}
```

Mit CopyTo in ein Array kopieren

Die **CopyTo**-Methode ist recht flexibel, da sie es erlaubt, sowohl die Anzahl als auch die Position der zu kopierenden Elemente aus dem **ArrayList**-Objekt sowie der zu ersetzenden Elemente aus dem Array festzulegen.

Variante 1 nur mit dem Zielarray als Parameter sieht so aus und ersetzt alle Elemente des Zielarrays durch die Elemente der **ArrayList**:

```
arrayList = new ArrayList();
arrayList.Add("Element 4");
arrayList.Add("Element 5");
arrayList.Add("Element 6");
String[] Zielarray = new String[3];
Zielarray[0] = "Element 1";
Zielarray[1] = "Element 2";
Zielarray[2] = "Element 3";
arrayList.CopyTo(Zielarray);
```

Die erste Überladung dieser Methode erwartet als zweiten Parameter den Index des Zielarrays, an dem mit dem Einfügen begonnen werden soll – zum Beispiel:

```
arrayList.CopyTo(Zielarray,1);
```

Dies kopiert den kompletten Inhalt von **arrayList** in **Zielarray**, und zwar dort ab dem mit dem zweiten Parameter angegebenen Index. Für die obige Konstellation würde dies einen Fehler auslösen, weil drei Elemente ab dem zweiten Element des Arrays nicht genügend Platz haben.

Die zweite Überladung erwartet als ersten Parameter den Startindex für das Zielarray, als dritten den Index des ersten zu kopierenden Elements und als vierten die Anzahl der von dieser Position aus zu kopierenden Elemente:

```
arrayList.CopyTo(1,Zielarray,1,1);
```

Mit der obigen Konstellation würde nur das Element **Element 5** an die Position von **Element 2** kopiert.

Steuerelemente anordnen

Unter Access ist das Anordnen von Steuerelementen einfach: Sie platzieren diese einfach auf dem Formular. In neueren Access-Versionen gibt es noch die Eigenschaften zum Verankern von Steuerelementen, aber damit sind die Möglichkeiten bereits ausgeschöpft. Unter WPF gibt es zahlreiche weitere Möglichkeiten, um die Position von Steuerelementen festzulegen. Dieser Artikel zeigt, wie es funktioniert und welche Vorteile sich gegenüber Access ergeben. Dabei lernen Sie auch gleich noch das Grid-Steuerelement und das GridSplitter-Steuerelement kennen, mit dem Sie Bereiche des Fensters vergrößern oder verkleinern können.

Das Ziel beim Programmieren mit WPF und einer Programmiersprache wie C# ist ja, die Benutzeroberfläche unabhängig vom dahinter steckenden Code entwickeln zu können – und umgekehrt. In der ersten Instanz werden wir noch nicht komplett dazu übergehen, sondern beispielsweise das Auslösen von Ereignissen noch in der Definition der Benutzeroberfläche verankern. Später lernen Sie Techniken kennen, auch dies von der Benutzeroberfläche zu entkoppeln. Theoretisch können Sie also tatsächlich einen Designer mit der Bearbeitung der Benutzeroberfläche betrauen, während Sie sich um den Code kümmern, der die Benutzeroberfläche steuert und von ihren Elementen ausgelöst wird.

Hierarchische Struktur

Wenn Sie ein Steuerelement zu einem neuen Fenster hinzufügen, sieht dies zunächst nicht viel anders aus wie eines in einem Access-Formular. Die XAML-Definition zeigt allerdings schnell, dass hier eine verschachtelte Struktur entstanden ist: Das **Window**-Element (das dem Formular unter Access entspricht) enthält zunächst ein **Grid**-Element, das dann wiederum etwa ein **Button**-Element aufnimmt (siehe Bild 1). Was es mit dem **Grid**-Element auf sich hat, schauen wir uns weiter unten an.

Element markieren

Wenn Sie nun das **Button**-Element markieren wollen, klicken Sie es einfach an. Beim **Grid**-Element und beim **Window**-Element wird das schon schwieriger. Nach etwas Experimentieren findet man heraus, dass man das **Window**-Element mit einem Klick auf den Fenstertitel markieren kann. Eine sichere Methode, das richtige Element zu markieren, bietet das Kontextmenü eines der Elemente. Dazu wählen Sie das Element der untersten gewünschten Hierarchie-Ebene aus, also beispielsweise das **Button**-Element aus der Abbildung. Das Kontextmenü bietet per Rechtsklick auf das Element den Eintrag **Aktuelle Aus-**

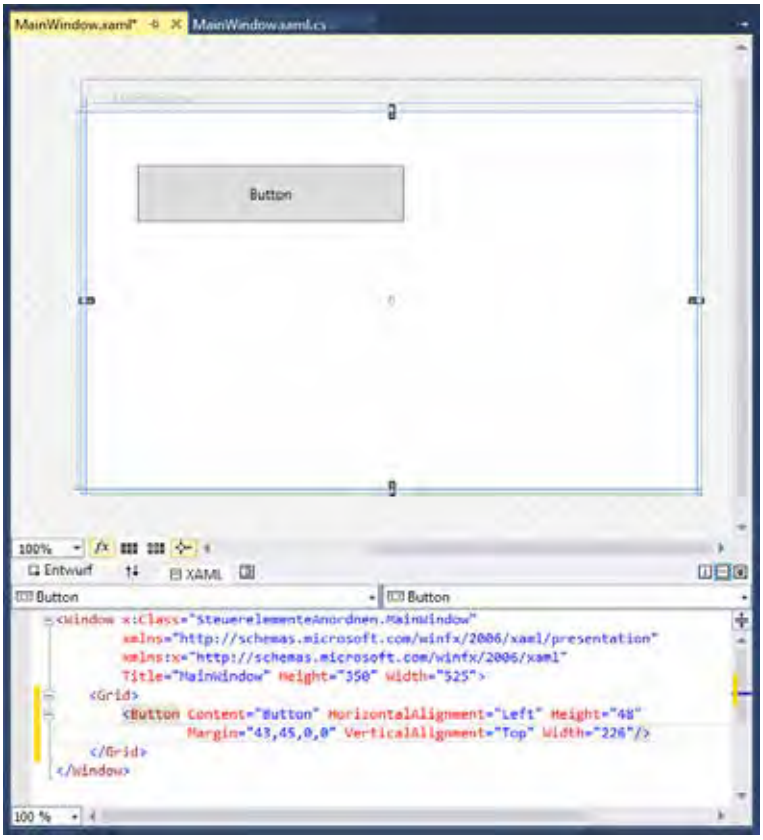


Bild 1: Ein Element mit seinen Eigenschaften

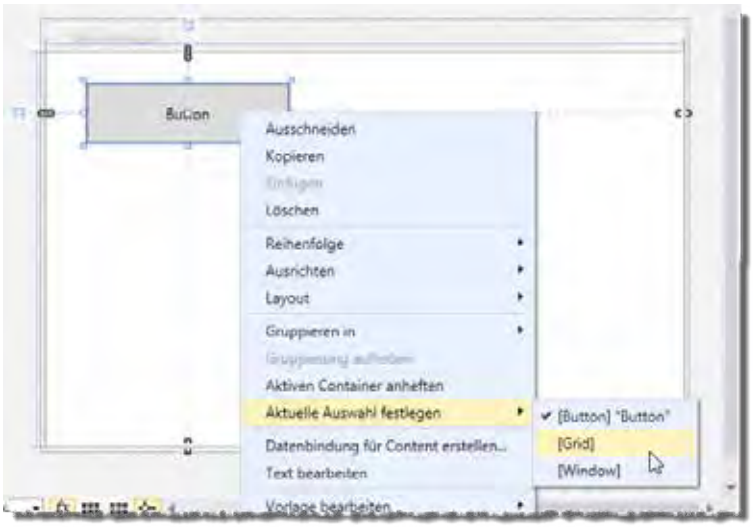


Bild 2: Markieren eines Elements der Hierarchie per Kontextmenü

wahl festlegen an, mit dem Sie eines der Elemente **[Button]**, **[Grid]** oder **[Window]** auswählen können (siehe Bild 2).

Größe und Position

Im Gegensatz zum Formularentwurf unter Access fällt gleich auf, dass die relevanten Maße des aktuell markierten Elements im Entwurf angezeigt werden. So werden die Abstände zum übergeordneten Element (im Falle des **Button**-Elements etwa zum **Grid**-Element) ständig angezeigt. Die Breite und Höhe eines Elements erscheinen, wenn Sie diese mit der Maus ändern (siehe Bild 3).

Width und Height

Die Größe finden Sie in der WPF-Definition unter den Eigenschaften **Height** beziehungsweise **Width** des jeweiligen Elements wieder. Damit legen Sie die Größe des Elements explizit fest. Mit **MinWidth, MaxWidth**, **MinHeight** und **MaxHeight** gibt es jedoch noch vier weitere Eigenschaften. Damit geben Sie die minimale und die maximale Höhe und Breite für ein Steuerelement an. Standardmäßig erscheint das Steuerelement dann mit den in **MinWidth** und **MinHeight** angegebenen Maßen. Es kann jedoch sein, dass sich der ursprüngliche Inhalt einmal ändert – beispielsweise, weil eine Schaltflächenbeschriftung in einer anderen Sprache angezeigt werden soll und mehr Platz einnimmt oder weil die Beschriftung/Schriftart/Schriftgröße je nach Kontext verändert werden soll. In einem solchen Fall würde man die **Min...**-Eigenschaften nutzen.

Sind beispielsweise **MinWidth**, **MaxWidth** und **Width** gesetzt, wird **Width** verwendet, sofern es innerhalb des Bereichs von **MinWidth** und **MaxWidth** liegt – das gilt auch für die entsprechenden **Height**-Werte.

Sind keine Breiten- oder Höhenangaben gemacht worden, richten sich die Abmessungen nach den maximal für den Inhalt nötigen Werten.

Wenn Sie die aktuelle Höhe und Breite eines Elements ermitteln wollen, verwenden Sie die Eigenschaften **ActualHeight** und **ActualWidth**.

Sie sollten keine absoluten Werte für Höhe und Breite angeben, wenn dies nicht dringend nötig ist – anderenfalls kann es sein, dass ein zur Laufzeit zugewiesener Text abgeschnitten dargestellt wird.

Margin

Die Position relativ zum übergeordneten Element ergibt sich aus der Eigenschaft **Margin**. Diese nimmt verschiedene Zeichenfolgen entgegen. Enthält diese etwa nur einen Zahlenwert, wird dieser für die Abstände zum linken, rechten,

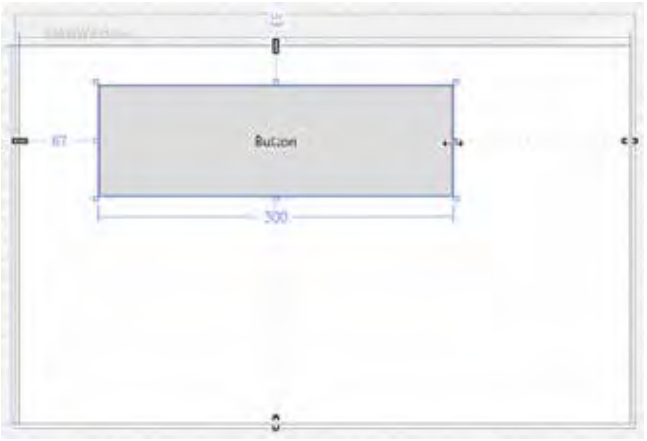


Bild 3: Anzeigen der Größe bei Anpassen von Höhe oder Breite

oberen und unteren Rand verwendet. Der folgende Button legt den Abstand von allen vier Seiten des übergeordneten Elements auf **10** fest:

```
<Button Margin="10" Width="96" Height="25" ... />
```

Wenn Sie diese vier Werte einzeln angeben möchten, können Sie einen Ausdruck für **Margin** angeben, der die vier Werte für **Left**, **Top**, **Width** und **Height** als kommaseparierte Liste entgegennimmt:

```
<Button Margin="10, 10, 10, 10" Width="96" Height="25" ... />
```

VerticalAlignment und HorizontalAlignment

Unter Access kannten Sie möglicherweise die mit den neueren Versionen eingeführten Eigenschaften **Horizontaler Anker** und **Vertikaler Anker**. Damit konnten Sie festlegen, ob ein Steuerelement beim Verändern der Formulargröße nach rechts oder unten verschoben wird oder ob sogar seine Größe entsprechend angepasst wurde.

Unter WPF übernehmen die beiden Eigenschaften **VerticalAlignment** und **HorizontalAlignment** diese Aufgabe. Die Eigenschaft **HorizontalAlignment** kann die Werte **Left**, **Center**, **Right** und **Stretch** annehmen, die Eigenschaft **VerticalAlignment** die Werte **Top**, **Center**, **Bottom** und **Stretch**. Bild 4 zeigt Beispiele für Schaltflächen, die mit den Werten **Left**, **Center** und **Right** für **HorizontalAlignment** und **Top**, **Center** und **Bottom** für **VerticalAlignment** ausgestattet wurden:

```
<Grid>
    <Button Content="Button" HorizontalAlignment="Left"
VerticalAlignment="Top" />
    <Button Content="Button" HorizontalAlignment="Left"
VerticalAlignment="Center" />
    <Button Content="Button" HorizontalAlignment="Left"
VerticalAlignment="Bottom" />
    <Button Content="Button" HorizontalAlignment="Center"
VerticalAlignment="Top" />
    <Button Content="Button" HorizontalAlignment="Center"
VerticalAlignment="Center" />
    <Button Content="Button" HorizontalAlignment="Center"
VerticalAlignment="Bottom" />
    <Button Content="Button" HorizontalAlignment="Right"
VerticalAlignment="Top" />
    <Button Content="Button" HorizontalAlignment="Right"
VerticalAlignment="Center" />
    <Button Content="Button" HorizontalAlignment="Right"
VerticalAlignment="Bottom" />
</Grid>
```

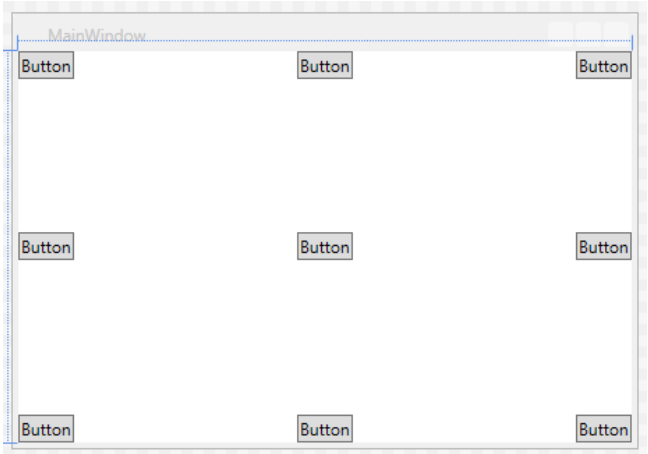


Bild 4: Schaltflächen mit verschiedenen Ausrichtungen

Mit diesen Einstellungen bleiben die **Button**-Elemente jeweils am Rand kleben, das mittlere in der Mitte.

Die hier nicht verwendete Einstellung ist gleichzeitig die Standardeinstellung: Wenn Sie **HorizontalAlignment** oder **VerticalAlignment** weglassen, verwendet WPF die Einstellung Stretch für das jeweilige Attribut.

Padding

Mit dem Attribut **Padding** legen Sie fest, welchen Abstand der Inhalt eines Steuerelements, also beispielsweise die Beschriftung einer Schaltfläche, von den Rändern haben soll. Mit **Padding="10,10,10,10"** legen Sie also den Abstand des Inhalts von den Rändern des Objekts fest.

Dabei hängt es von den Einstellungen der beiden Eigenschaft **HorizontalContentAlignment** und **VerticalContentAlignment** ab, ob die Einstellungen für **Padding** tatsächlich

relevant sind. Bei einer großen Schaltfläche, deren Beschriftung klein ist und wie folgt ausgerichtet wird, spielt Padding keine Rolle:

```
<Button Content="Button mit einer etwas umfangreicheren
Beschriftung" HorizontalAlignment="Left" Height="50" Mar-
gin="50,50,0,0" VerticalAlignment="Top" Width="200"
                Padding="10,10,10,10" HorizontalContentAli
gnment="Center" VerticalContentAlignment="Center"/>
```

Standardeinstellungen

Neu hinzugefügte WPF-Elemente erhalten je nach der Art des Hinzufügens unterschiedliche Standardattribute. Wenn Sie etwa einfach auf das **Button**-Element im Werkzeugkasten klicken und dann im Fenster per Klick die Position markieren, an der das Steuerelement landen soll, fügt Visual Studio mit **Content** die Beschriftung **Button**, mit **HorizontalAlignment** und **VerticalAlignment** die Werte **Left** und **Top**, mit **Margin** die aktuelle Position und mit **Width** eine Standardbreite von **75** vor.

Was aber geschieht, wenn wir alle Attribute außer **Content** entfernen? In diesem Fall nimmt das **Button**-Steuerelement die komplette Fläche des übergeordneten Elements ein. Das bedeutet also, dass **HorizontalAlignment** und **VerticalAlignment** offensichtlich den Wert **Stretch** als Standardwert einsetzen.

Experimente zum Einüben der Eigenschaften

Am besten finden Sie sich in die verschiedenen Eigenschaften für die Einstellung der Größe und Ausrichtung ein, indem Sie ein wenig experimentieren. Das gelingt in diesem Fall sehr gut, wenn Sie sich auf die Eingabe per XAML konzentrieren. Legen Sie beispielsweise in einem leeren Fenster einmal folgenden Code an:

```
<Window ... Title="MainWindow" Height="350" Width="525">
    <Grid>
        <Button Content="Beispielschaltfläche" />
    </Grid>
</Window>
```

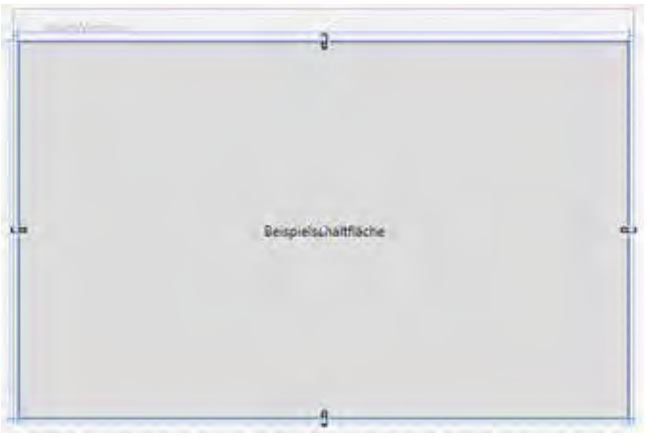


Bild 5: Schaltfläche ohne Attribute für Größe und Position

Dies zeigt eine Schaltfläche an, die sich über das komplette **Grid**-Element im Fenster erstreckt (siehe Bild 5).

Ergänzen Sie nun das Attribut **Margin** wie folgt, bringt dies einen Abstand zum oberen, linken, rechten und unteren Rand wie in Bild 6 ein:

```
<Button Content="Beispielschaltfläche"
    Margin="20,20,20,20"/>
```

Stellen Sie dann die Attribute **HorizontalAlignment** und **VerticalAlignment** auf **Left** beziehungsweise **Top** ein, werden die Eigenschaften für den rechten und unteren Rand im Attribut **Margin** wirkungslos (siehe Bild 7):

```
<Button Content="Beispielschaltfläche"
    Margin="20,20,20,20" HorizontalAlignment="Left"
    VerticalAlignment="Top"/>
```

Bild 6: Schaltfläche mit **Margin**

Bild 7: Schaltfläche mit horizontaler und senkrechter Ausrichtung

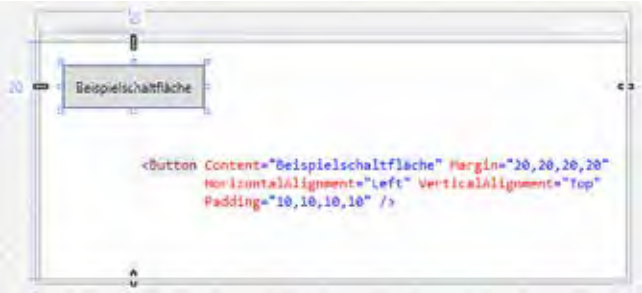


Bild 8: Schaltfläche mit Padding

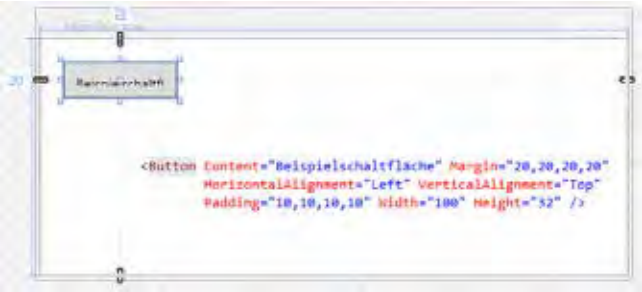


Bild 9: Schaltfläche mit Padding und zu wenig Platz für die Beschriftung

Nun wollen wir, dass die Schaltfläche etwas luftiger aussieht und vergrößern den Abstand vom Inhalt, also der Beschriftung, zum Rand der Schaltfläche. Dies erledigen wir mit der Eigenschaft **Padding** (Ergebnis siehe Bild 8).

```
<Button Content="Beispielschaltfläche"
    Margin="20,20,20,20" HorizontalAlignment="Left"
    VerticalAlignment="Top" Padding="10,10,10,10" />
```

In Bild 9 sehen Sie schließlich noch, warum eine feste Höhe und Breite sich nachteilig auswirken kann: Durch die **Padding**-Eigenschaft und den dadurch entstehenden Abstand zum Steuerelementrand wird der Platz für die Beschriftung zu eng:

```
<Button Content="Beispielschaltfläche" Margin="20,20,20,20"
    HorizontalAlignment="Left" VerticalAlignment="Top"
    Padding="10,10,10,10" Width="100" Height="32" />
```

Das Grid-Element

Es gibt unter WPF verschiedene Elemente zum Positionieren von Steuerelementen – sogenannte **Panel**-Elemente. Eines davon heißt **Grid**. Dieses haben Sie weiter oben bereits kennen gelernt. Es wird standardmäßig zu neuen Fenstern hinzugefügt. So, wie wir es oben genutzt haben, macht es sich nicht bemerkbar, denn es besitzt nur eine einzige Zelle. Mit der Bezeichnung »Zelle« gelangen wir auch direkt zur prinzipiellen Bauweise eines solchen Grids: Es hat nämlich starke Ähnlichkeit mit einer Tabelle, die ihre Inhalte durch Zeilen und Spalten unterteilt.

Das **Grid**-Element bietet interessanterweise keine Attribute, mit denen Sie etwa die Anzahl der Zeilen und Spalten angeben können. Stattdessen definieren Sie die Struktur über zwei Unterelemente namens **Grid.RowDefinitions** und **Grid.ColumnDefinitions**. Darin legen Sie für jede Zeile ein weiteres Element namens **RowDefinition** beziehungsweise **ColumnDefinition** ab. Das folgende Beispiel ändert zunächst nichts, da wir hier ja nur eine Zeile und eine Spalte definieren:

```
<Grid >
    <Grid.RowDefinitions>
        <RowDefinition />
    </Grid.RowDefinitions>
    <Grid.ColumnDefinitions>
        <ColumnDefinition />
    </Grid.ColumnDefinitions>
</Grid>
```

Um eine sichtbare Wirkung zu erzielen, müssten wir also zumindest ein weiteres **RowDefinition**- oder **ColumnDefinition**-Element hinzufügen. Dies erledigen wir gleich:

Daten im DataGrid-Steuerelement anzeigen I

In einem Access-Magazin würde man erstmal die Schaltfläche oder das Textfeld als Steuerelemente vorstellen. Im DATENBANKENTWICKLER wollen wir jedoch von mehreren Seiten an die Programmierung von Datenbank-Anwendungen mit Visual Studio herangehen – unter anderem, indem wir die Nutzung von ADO.NET vorstellen oder die Erstellung von Benutzeroberflächen mit WPF beschreiben. Für beides wollen wir überhaupt einmal die Daten der gewünschten Datenherkunft anzeigen, statt diese immer per Console auszugeben. Dies gelingt am einfachsten mit dem DataGrid-Steuerelement, das stark vereinfacht mit einem Unterformular unter Access in der Datenblattansicht zu vergleichen ist.

Das **DataGrid**-Steuerelement finden Sie, wenn Sie ein Projekt für C#/WPF erstellt haben, direkt in den **Häufig verwendten WPF-Steuerelementen** im **Werkzeugkasten**. Ziehen Sie dieses einfach in den Entwurf eines Fensters und benennen Sie es über die Eigenschaft beispielsweise **ctlDataGrid**. Nun benötigen Sie eine Datenquelle. Dafür verwenden wir die Beispieldatenbank **Suedsturm.mdb**, die wir einfach als Ressource zum Projekt hinzufügen. Dazu ziehen Sie die entsprechende Datei einfach auf das Projekt im Projektmappen-Explorer und stellen dann die Eigenschaft **In Ausgabeverzeichnis kopieren** auf den entsprechenden Wert ein (siehe Bild 1):

- **Nicht kopieren**: Kopiert die Datei niemals in das Ausgabeverzeichnis.
- **Immer kopieren**: Kopiert die Datei immer beim Erstellen der Anwendung in das Ausgabeverzeichnis.
- **Kopieren, wenn neuer**: Kopiert die Datei nur in das Ausgabeverzeichnis, wenn die aktuell im Projekt gespeicherte Version neuer als die Version im Ausgabeverzeichnis ist.

Wenn Sie immer eine unbefleckte Version der Datei benötigen, was beispielsweise bei Tests vorteilhaft ist, wählen Sie **Immer kopieren**. Wenn Sie hingegen in der Anwendung Daten ändern und beim nächsten Erstellen wieder auf die geänderten Daten zugreifen möchten, wählen Sie **Kopieren, wenn neuer**.

Damit das **DataGrid**-Steuerelement nun etwa die Daten einiger Felder der Tabelle **tblArtikel** anzeigt, müssen Sie an

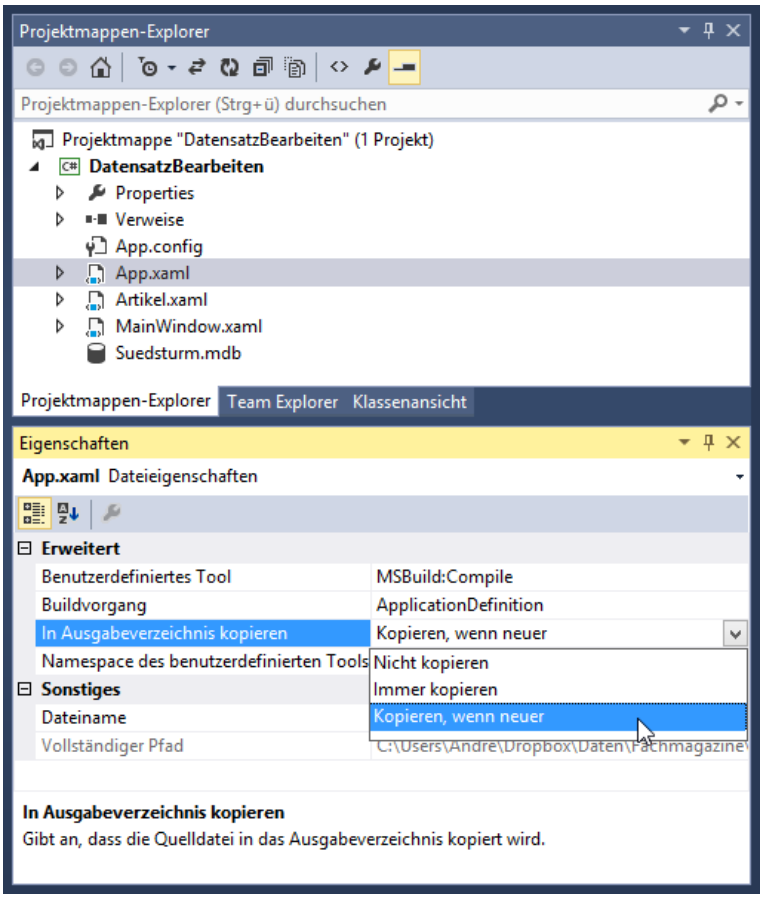


Bild 1: Hinzufügen einer Datenbankdatei zu einem Projekt

den Eigenschaften des **DataGrid**-Steuerelements erstmal nichts ändern. Wir wollen zu Beispielzwecken ein neues Fenster namens **Artikel** erstellen, was wir mit dem Menübefehl **Projekt|Fenster hinzufügen** erledigen. Diesem fügen Sie dann wie in Bild 2 ein **DataGrid**-Steuerelement hinzu.

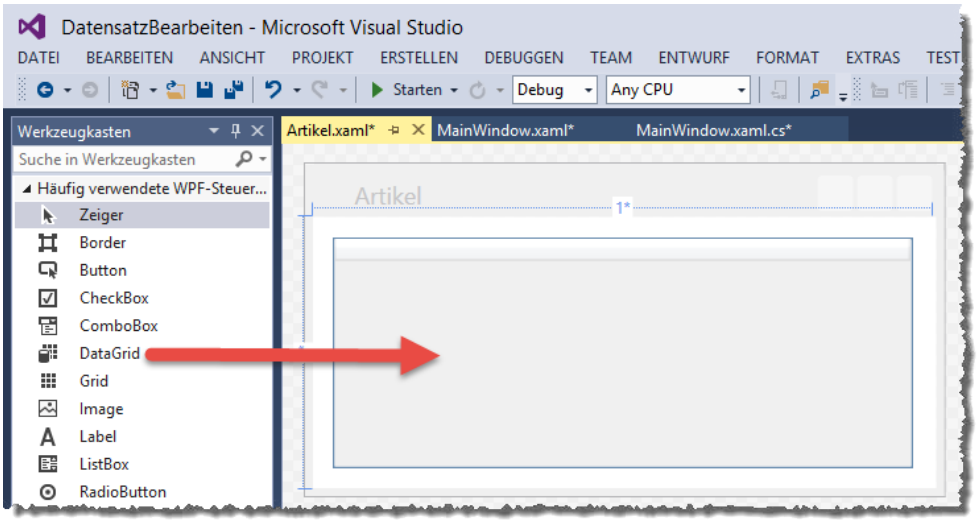


Bild 2: Hinzufügen des **DataGrid**-Steuerelements

Fenster aufrufen

Beim Starten des Projekts mit dem entsprechenden Menübefehl zeigt dieses automatisch das Fenster **MainWindow.xaml** an. Diesem fügen wir nun eine Schaltfläche hinzu, mit der Sie unser Beispielfenster **Artikel.xaml** öffnen können.

Hinterlegen Sie für diese Schaltfläche namens **btnArtikelAnzeigen** die folgende Ereignisprozedur (Schaltfläche markieren, dann im Eigenschaftsfenster auf den Blitz klicken, dann Doppelklick in das Eigenschaftsfeld **Click** – das Ergebnis sieht wie in Bild 3 aus):

```
private void btnArtikelAnzeigen_Click(object sender,
        RoutedEventArgs e) {
    Artikel wnd = new Artikel();
    wnd.Show();
}
```

Diese Prozedur erzeugt in der Variablen **wnd** eine neue Instanz des Fenster-Objekts und blendet dieses mit der **Show**-Methode ein. Ein Klick auf Starten öffnet das Fenster **MainWindow**, von dem aus Sie mit der Schaltfläche **Artikel anzeigen** das gewünschte Fenster mit dem **DataGrid**-Steuerelement öffnen können. Dieses zeigt erwartungsgemäß zunächst keine Daten an.

Einfache Daten anzeigen

Das ändert sich allerdings, wenn wir dem Fenster **Artikel.xaml** eine Schaltfläche namens **btnArtikelInDataGrid** hinzufügen, welche die Ereignisprozedur aus Listing 1 ausführt.

Dies liefert im Fenster **Artikel.xaml** die Ansicht aus Bild 4. Damit die Prozedur funktioniert, benötigen Sie zwei Klassen,

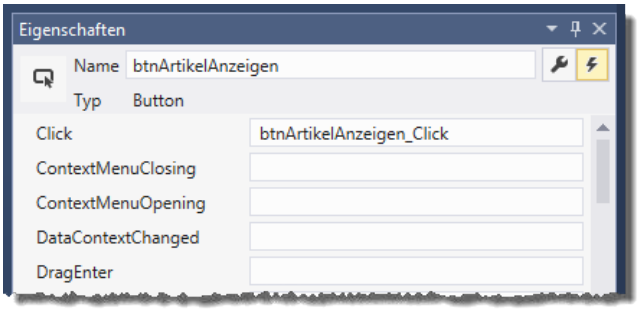


Bild 3: Ereignisprozedur anlegen

Bild 4: Erster Testlauf

```
private void btnArtikelInDataGrid_Click_1(object sender, RoutedEventArgs e)
{
    string Connectionstring = "Provider=Microsoft.Jet.OLEDB.4.0;Data Source=Suedsturm.mdb";
    OleDbConnection cnn = new OleDbConnection(Connectionstring);
    OleDbCommand cmd = new OleDbCommand();
    cmd.Connection = cnn;
    cmd.CommandText = "SELECT ArtikelID, Artikelname FROM tblArtikel";
    cnn.Open();
    OleDbDataReader dr = cmd.ExecuteReader();
    DataTable dt = new DataTable();
    dt.Load(dr);
    ctlDataGrid.ItemsSource = dt.DefaultView;
    cnn.Close();
}
```

Listing 1: Füllen eines **DataGrid**-Steuerelements mit den Werten einer Tabelle

die Sie mit der **using**-Anweisung im Kopf des Klassenmoduls **Artikel.xaml.cs** einbinden:

```
using System.Data;
using System.Data.OleDb;
```

Die Erste liefert das hier benötigte **DataTable**-Objekt, die Zweite einige weitere Objekte, die speziell für den Zugriff auf eine Access-Datenbank geeignet sind. Wenn Sie hier statt auf eine Access-Datenbank etwa auf eine SQL Server-Datenbank zugreifen wollten, müssten Sie beispielsweise die Klasse **System.Data.Sqlclient** einbinden und später die entsprechenden Objekte verwenden.

Hier aber geht die Prozedur wie folgt vor: Zunächst trägt sie die Verbindungszeichenfolge, welche den Provider und die Datenquelle definiert, in die Variable **Connectionstring** ein. Da die Datei **Suedsturm.mdb** im Ausgabeverzeichnis der Anwendung landet, brauchen wir hier keinen Pfad anzugeben. Danach erstellt die Prozedur für die Variable **cnn** ein neues **OleDbConnection**-Objekt und übergibt beim Erstellen mit dem **new**-Schlüsselwort die Verbindungszeichenfolge aus **Connectionstring**. Für VBA-Entwickler ist dies Neuland: Dort wurde beim Erstellen niemals ein Parameter übergeben. Unter C# und anderen .NET-Sprachen ist dies jedoch möglich. Dann erstellt die Prozedur ein neues **OleDbCommand**-

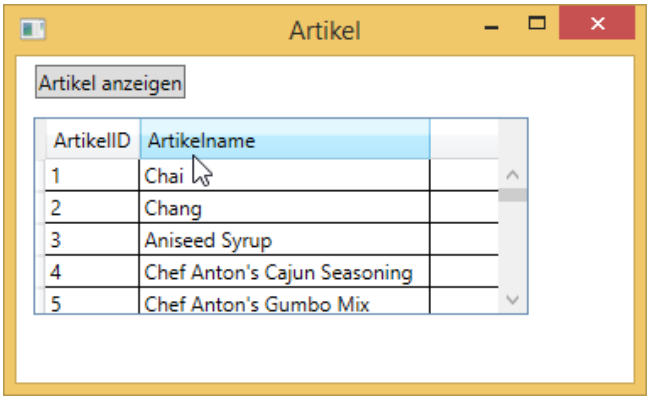


Bild 5: Anzeige der Daten der Tabelle **tblArtikel**

Objekt namens **cmd**, diesmal ohne Parameter. Das Objekt aus **cnn** weisen wir dann der Eigenschaft **Connection** des **Command**-Objekts in **cmd** zu. Die SQL-Anweisung, welche die anzuzeigenden Daten liefern soll, landet hingegen in der Eigenschaft **CommandText**: **SELECT ArtikelID, Artikelname FROM tblArtikel**.

Die **Open**-Methode des Objekts **cnn** öffnet die Verbindung, die **ExecuteReader**-Methode des **Command**-Objekts erstellt ein neues **OleDbDataReader**-Objekt und speichert es in der Variablen **dr**.

Den Inhalt des **OleDbDataReader**-Objekts können wir leider nicht direkt dem **DataGrid**-Steuerelement zuweisen. Dazu benötigen wir noch ein weiteres Objekt des Typs **DataTab-**

Objektorientierte Programmierung: Grundlagen II

Im ersten Teil dieser Artikelreihe haben wir uns die Grundlagen zur objektorientierten Programmierung mit C# angesehen – zum Beispiel die Klasse Program, das Erstellen neuer Klassen, die Verwendung von Eigenschaften und von Methoden. Dies greifen wir im zweiten Teil auf und gehen auf verschiedene Arten der Eigenschaftsdefinition, Konstruktoren, aufgeteilte Klassen, Objektreferenzen, Überladung von Methoden und verschiedene Parameter-Techniken ein.

Öffentliche Variablen oder »Felder«

Wenn eine Eigenschaft einer Klasse nur gelesen und geschrieben werden soll, ohne dass beim Lesen oder Schreiben weitere Aktionen erfolgen sollen, können Sie diese theoretisch einfach als öffentliche Variable deklarieren:

```
class Person_VornameOeffentlich
{
    public string Vorname;
}
```

Sie greifen dann lesend und schreibend auf die Variable zu:

```
public void OeffentlicheVariableTesten()
{
    Person_VornameOeffentlich person =
        new Person_VornameOeffentlich();
    person.Vorname = "André";
    Console.WriteLine("Vorname: {0}", person.Vorname);
}
```

Eine solche Variable/Eigenschaft nennt man auch Feld.

Eigenschaft kapseln

Meist ist aber kein derart unkontrollierter Zugriff auf eine solche Variable erwünscht. Dann deklariert man die Variable als private Variable und erstellt eine öffentliche Eigenschaft, die einen **get**- und einen **set**-Block enthält. In der Kurzform sieht dies so aus:

```
private string vorname;
```

Dies ist die öffentliche Variable mit dem **get**- und dem **set**-Block – **get** liefert den aktuellen Wert der privaten Variablen **vorname** aus und **set** stellt diesen beim Zuweisen eines Wertes für die Eigenschaft auf den Inhalt der Variablen **value** ein:

```
public string Vorname
{
    get { return vorname; }
    set { vorname = value; }
}
```

Damit halten Sie sich die Möglichkeit offen, den Wert der Eigenschaft auch innerhalb der Klasse für andere Zwecke einzusetzen – etwa für Berechnungen weiterer Eigenschaftswerte oder zum Zusammensetzen von Ausdrücken. Gegebenenfalls benötigen Sie dies gar nicht und wollen nur von außen den Wert der Eigenschaft eintragen und diesen später wieder auslesen.

Ein weiterer Vorteil der Kapselung ist: Sie können damit auch festlegen, dass eine Eigenschaft nur schreibend oder nur lesend genutzt werden kann. Wenn die Eigenschaft Vorname beispielsweise nur lesend genutzt werden soll, lassen Sie einfach den **set**-Teil weg:

```
public string Vorname
{
    get { return vorname; }
}
```

Für einen ausschließlich schreibenden Zugriff bleibt hingegen der **get**-Teil außen vor:

```
public string Vorname
{
    set { vorname = value; }
}
```

Getter und Setter schnell erstellen

Benötigen Sie **get**- und **set**-Block, wollen aber keine Einschränkungen vornehmen wie etwa eine Prüfung der Eingabe und benötigen auch keine Variable, auf die Sie von der Klasse aus zugreifen wollen, können Sie diese verkürzte Variante nutzen:

```
class Person_AutomatischImplementiert
{
    public string Vorname { get; set; }
    public string Nachname { get; set; }
}
```

Der Zugriff auf die beiden Eigenschaften der Klasse erfolgt dann genau wie bei der vorherigen Version.

Unterstützung durch Visual Studio

Beim schnellern Erstellen von Eigenschaften unterstützt Visual Studio Sie mit Shortcuts. Dazu geben Sie in einer leeren Zeile in der gewünschten Klasse zunächst die vier Buchstaben **prop** ein. Nun erscheinen bereits weiterführende Möglichkeiten in der IntelliSense-Liste (siehe Bild 1). Für uns sind aktuell zwei Varianten interessant. Die erste erfordert die Eingabe von **prop + Tab + Tab** und liefert die folgende Vorlage:

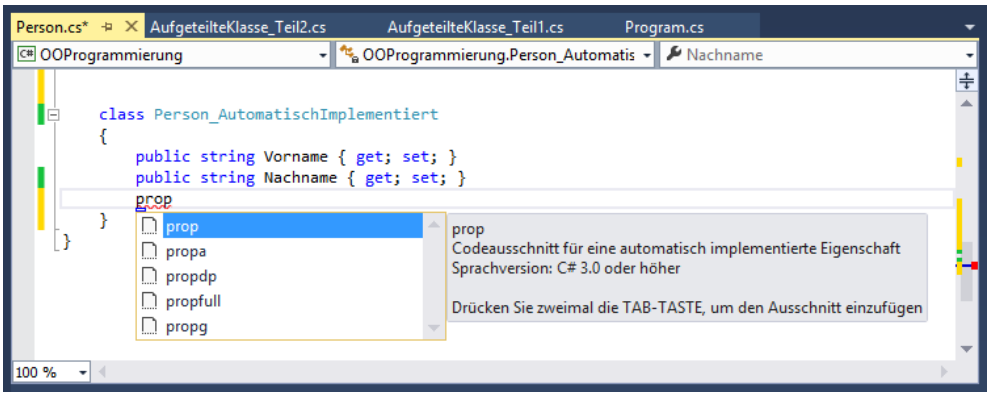


Bild 1: Schnelles Erstellen einer Property

```
public int MyProperty { get; set; }
```

Dabei werden int und **MyProperty** gelb hinterlegt und können mit der Tabulator-Taste durchlaufen werden. Auf diese Weise können Sie die Platzhalter leicht durch den Datentyp und den Namen der Eigenschaft ersetzen.

Die zweite Variante ist die Eingabe von **propfull + Tab + Tab** (es reicht auch **propf + Tab + Tab** aus). Dies liefert die folgende Vorlage:

```
private int myVar;

public int MyProperty
{
    get { return myVar; }
    set { myVar = value; }
}
```

Hier können Sie den Datentyp (**int**), den Variablennamen (**myVar**) und den Eigenschaftsnamen (**MyProperty**) per Tabulatortaste erreichen und ändern.

Klassen mit Konstruktor

Unter VBA gibt es keine Möglichkeit, gleich beim Erstellen einer Klasse einen oder mehrere Werte zu übergeben. Dazu muss man nach dem Erstellen dafür vorgesehene Eigenschaften nutzen. Das ist schon ein Nachteil, wenn für das Verwenden der Klasse bestimmte Pflichtinformationen übergeben werden müssen, bevor man etwa bestimmte Methoden dieser Klasse nutzt.

Das heißt, dass man vor dem Ausführen der Methode immer noch prüfen muss, ob die Pflichtinformationen überhaupt übergeben wurden. Davon abgesehen ist es natürlich auch viel einfacher,

die wichtigsten Eigenschaften gleich beim Initialisieren des Objekts zu übergeben statt mühsam Eigenschaft für Eigenschaft zu übergeben.

Unter C# und anderen objektorientierten Programmiersprachen gibt es deshalb die sogenannten Konstruktoren. Das sind Parameter, die gleich beim Erstellen einer Klasse übergeben werden müssen beziehungsweise können.

Bevor Sie eine Klasse mit einem Konstruktor erstellen, müssen Sie sich überlegen, welche Varianten Sie für die Initialisierung mit oder ohne Parameter zur Verfügung stellen wollen. Eine Klasse **Person** mit den beiden Eigenschaften **Vorname** und **Nachname** kann beispielsweise diese beiden Eigenschaften beim Initialisieren per Parameter entgegennehmen.

Die Definition der Klasse sieht dann etwa wie folgt aus:

```
class Person {
    private string vorname;
    private string nachname;
    public Person() {}
    public Person(string Vorname,
            string Nachname) {
        vorname = Vorname;
        nachname = Nachname;
    }
    public string Vorname {
        get { return vorname; }
        set { vorname = value; }
    }
    public string Nachname {
        get { return nachname; }
        set { nachname = value; }
    }
```

Hier sehen Sie die beiden privat deklarierten Variablen **vorname** und **nachname**. Außerdem sind natürlich die beiden Eigenschaftsmethoden **Vorname** und **Nachname** mit den **get**- und **set**-Blöcken vorhanden.

Neu ist zunächst die folgende Zeile:

```
public Person() {}
```

Dies ist der erste Konstruktor. Er ermöglicht die Initialisierung eines Objekts auf der Basis der Klasse **Person**, ohne direkt eine der beiden Eigenschaften **Vorname** oder **Nachname** übergeben zu müssen. Der Konstruktor hat genau den gleichen Namen wie die Klasse selbst. Der Unterschied zu einer normalen öffentlichen Variablen ist, dass Sie hinter dem Bezeichner erstens ein rundes Klammernpaar (**()**) und dann noch ein geschweiftes Klammernpaar finden (**{}**).

Der zweite Konstruktor erwartet die Übergabe von Werten für die beiden Parameter **Vorname** und **Nachname**. Er heißt ebenfalls genauso wie die Klasse, enthält aber in Klammern eine Liste der Parameter, die beim Initialisieren übergeben werden müssen. Dahinter folgt ein Paar geschweifter Klammern, innerhalb derer die übergebenen Parameter **Vorname** und **Nachname** den Variablen **vorname** und **nachname** übergeben werden:

```
public Person(string Vorname, string Nachname) {
    vorname = Vorname;
    nachname = Nachname;
}
```

Die Eigenschaften der Klasse können Sie nun gleich beim Initialisieren übergeben:

```
Person person;
person = new Person("André", "Minhorst");
Console.WriteLine("{0} {1}", person.Vorname,
    person.Nachname);
//Ausgabe:
//André Minhorst
```

Wir können das Objekt aber, da wir ja auch einen parameterlosen Konstruktor angelegt haben, auch ohne Übergabe von Parametern initialisieren und **Vorname** und **Nachname** erst später übergeben.

Wenn es Pflichteigenschaften gibt, ohne welche die Methoden einer Klasse nicht sinnvoll genutzt werden können, können Sie natürlich auch entscheiden, dass die Eigenschaften auf jeden Fall direkt beim Initialisieren des Objekts gefüllt werden müssen. Dann legen Sie einfach nur einen Konstruktor an, der alle notwendigen Eigenschaften als Parameter erwartet und lassen den parameterlosen Konstruktor weg.

Von Class_Initialize zum Konstruktor

Für alle, die von VBA kommen: Unter VBA gibt es in Klassen die beiden Ereignisprozeduren **Class_Initialize** und **Class_Terminate**. Diese werden, soweit implementiert, beim Initialisieren und beim Zerstören der Klasse ausgelöst – also beim Setzen der letzten Objektvariablen mit einer Referenz auf die Klasse auf den Wert **Nothing**.

Sie raten richtig: Der beziehungsweise die Konstruktoren ersetzen das Ereignis **Initialize** (zum **Terminate**-Ereignis kommen wir später). Das heißt also, dass Sie im Konstruktor auch solche Anweisungen unterbringen sollten, die immer beim Initialisieren der Klasse ausgeführt werden sollen – beispielsweise das Einlesen von Informationen aus Datenbanken, das Erstellen weiterer Objekte et cetera. Wenn eine Klasse keinen Konstruktor benötigt, der beim Initialisieren Parameter zur Verfügung stellt, aber zu diesem Zeitpunkt dennoch Anweisungen ausführen wollen, verwenden Sie den oben bereits vorgestellten Konstruktor ohne Parameter:

```
public Person() {}
```

Diesem können Sie natürlich beliebige Anweisungen hinzufügen:

```
public Person() {
    //... wichtige Anweisungen beim Initialisieren
}
```

Umleitung eines Konstruktoraufrufs

Stellen wir uns nun vor, wir möchten dem Benutzer auch die Möglichkeit geben, nur den Nachnamen der Person beim Initialisieren an die Klasse **Person** zu übergeben. Dazu fügen wir einen weiteren Konstruktor zur Klasse hinzu, der nur diesen einen Parameter verwendet:

```
public Person_KonstruktorUmleiten(string Nachname)
{
    nachname = Nachname;
}
```

Damit haben wir nun eine Möglichkeit mehr, die Klasse zu initialisieren:

```
Person person = new Person("Minhorst");
```

Was hat das nun mit der Umleitung eines Konstruktoraufrufs zu tun? Nun: wir kommen hier wieder auf die Funktion des Konstruktors als die beim Starten ausgelöste Methode zurück, in der Sie Anweisungen zur Vorbereitung des Objekts unterbringen. Diese möchten Sie sicherlich nur in einem Konstruktor unterbringen und nicht in jedem einzelnen. Dazu wählen Sie dann logischerweise den Konstruktor aus, der die wenigsten Parameter entgegennimmt (im Optimalfall gar keinen).

Verwendet die initialisierende Klasse diesen Konstruktor, läuft alles nach Plan. Aber was geschieht, wenn einer der Konstruktoren genutzt wird, der einen oder mehrere Parameter verwendet? Es würde naheliegen, von diesem aus dann den Konstruktor mit den Anweisungen zu starten, die bei jeder Initialisierung aufgerufen werden sollen.

Die Vorgehensweise sieht jedoch etwas anders aus. Der Konstruktor enthält nur die Anweisungen, die bei jedem Initialisierungsvorgang ausgeführt werden sollen:

```
public Person() {
    //... Anweisungen beim Initialisieren
}
```

Der Konstruktor, der nur einen Parameter erwartet, trägt diesen wie gewohnt in die entsprechende Variable ein. Zusätzlich enthält der Kopf der Methode hinter einem Doppelpunkt

das Schlüsselwort **this()**, was bedeutet, dass es auf den Konstruktor ohne Parameter umgeleitet wird:

```
public Person(string Nachname) : this() {
    nachname = Nachname;
}
```

Ein Konstruktor mit zwei Parametern würde dann im Kopf auf den Konstruktor mit dem einen Parameter umleiten – was man an der Angabe des entsprechenden Parameters erkennen kann:

```
public Person(string Vorname,
        string Nachname) : this(Nachname) {
    vorname = Vorname;
}
```

Wenn Sie die Klasse nun mit dem Konstruktor mit einem Parameter initialisieren, löst dies nicht etwa zuerst die Anweisungen dieses Konstruktors aus, sondern die des hinter dem Doppelpunkt angegebenen Konstruktors. Beim Aufruf des Konstruktors mit den zwei Parametern ruft dieser erst den Konstruktor mit dem einen Parameter auf, der dann den Konstruktor ohne Parameter ansteuert. Sprich: Die Anweisungen innerhalb der Konstruktoren werden in umgekehrter Reihenfolge ausgeführt (also: Konstruktor ohne Parameter – Konstruktor mit einem Parameter – Konstruktor mit zwei Parametern).

Unterschied zwischen Konstruktor vs. Eigenschaften und Methoden

Wie kann man nun die Definition eines Konstruktors von der einer Methode oder einer Eigenschaft unterscheiden? Ein Konstruktor enthält niemals einen Datentyp für einen Rückgabewert (**int**, **string** et cetera) oder das Schlüsselwort **void** wie eine Methode ohne Rückgabewert.

Eigenschaften beim Initialisieren übergeben – auch ohne Konstruktor!

Wenn Sie die Eigenschaften einer Klasse bislang immer erst nach dem Initialisieren übergeben haben, möchten Sie nun vielleicht der einen oder anderen Klasse einen Konstruktor hinzufügen, um die Eigenschaften komfortabler zu übergeben. Dies ist jedoch gar nicht unbedingt nötig, denn es gibt noch eine andere Methode, die Eigenschaften gleich in der Zeile mit der **new**-Anweisung unterzubringen.

Dabei tragen Sie hinter dem Paar runder Klammern des Klassennamens eine öffnende geschweifte Klammer ein und fügen dort einfach entsprechende Name-Wert-Paare für die Eigenschaftswerte hinzu – also etwa wie folgt:

```
Person person;
person = new Person() {Vorname="André",
    Nachname="Minhorst"};
```

Mit der folgenden Anweisung greifen Sie dann wie gewohnt auf die Eigenschaften zu:

```
Console.WriteLine("{0} {1}", person.Vorname,
    person.Nachname);
```

Für mehr Komfort betätigen Sie nach der Eingabe der öffnenden geschweiften Klammern einmal die Tastenkombination **Strg + Leertaste**. Das Resultat finden Sie in Bild 2 – alle verfügbaren Eigenschaften werden per IntelliSense angezeigt.

Der Destruktor

Natürlich gibt es auch eine Möglichkeit, Anweisungen beim Zerstören des Objekts auszulösen – und zwar im sogenannten Destruktor. Die Anwendung eines Destruktors und das kontrollierte Zerstören eines Objekts inklusive Freigabe der darin benutzten Ressourcen würde jedoch an dieser Stelle den Rahmen sprengen. Daher gehen wir zu einem späteren Zeitpunkt auf dieses Thema ein.

Klassendefinitionen splitten

Normalerweise sollten Sie der Übersichtlichkeit halber für jede Klasse eine eigene Datei mit der Endung **.cs** erstellen. Sie können natürlich auch mehrere Klassen in einer Klassendatei unterbringen, wie wir es auch sicher in dem einen oder

Objektorientierte Programmierung: Vererbung I

Eines der wichtigsten Merkmale der objektorientierten Programmierung ist die Vererbung. Dank der Vererbung kann man für ein Objekt eine Basisklasse erstellen, die grundlegende Eigenschaften und Methoden für dieses Objekt bereitstellt. Benötigen Sie nun ein weiteres Objekt, das auf dem ersten Objekt aufbaut, aber weitere oder geänderte Eigenschaften und Methoden enthält, müssten Sie ohne Vererbung die gleiche Klasse nochmal erstellen und nach ihren Wünschen anpassen. Dank Vererbung brauchen Sie in der Klasse für das neue Objekt jedoch nur die Erweiterungen und Änderungen zu definieren. Den Rest verwenden Sie von der ersten Klasse – und brauchen eine Menge Code nur an einer Stelle zu warten.

Sie können natürlich, genau wie bei Tabellen einer Datenbank, ein Objekt erstellen, das alle denkbaren Eigenschaften und Methoden liefert (beziehungsweise im Fall einer Tabelle entsprechende Felder enthält). Ein Beispiel wäre eine Tabelle, die zunächst nur Personendaten enthält, die dann aber um Daten für verschiedene Arten von Personen erweitert wird – für Mitarbeiter, Kunden et cetera.

In der Tabelle blieben dann, je nachdem, ob man einen Mitarbeiter oder einen Kunden pflegen möchte, die jeweils für den anderen Personentyp vorgesehenen Felder leer. Damit diese nicht versehentlich gefüllt werden, würde man dafür verschiedene Formulare vorsehen, von denen eines nur die Basisdaten plus die Mitarbeiterfelder und ein anderes nur die Basisdaten plus die Kundenfelder anzeigt.

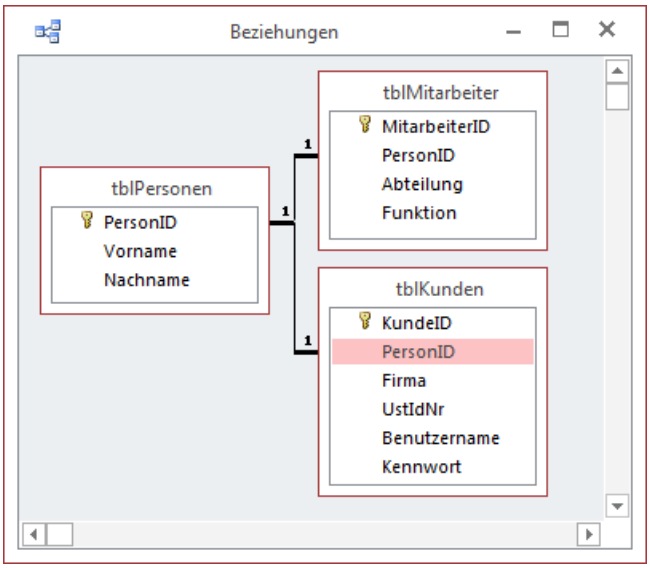


Bild 1: Spezialisierung in einem Datenmodell

Im Datenbankentwurf würde man hier nun eine Tabelle namens **tblPersonen** erstellen und für die jeweiligen Spezialisierungen Tabellen wie **tblKunden** oder **tblMitarbeiter**. Diese würden dann per 1:1-Beziehung miteinander verknüpft werden, damit nicht etwa die Daten eines Datensatzes der Tabelle **tblKunden** mehreren Datensätzen der Tabelle **tblPersonen** zugeordnet werden können und umgekehrt. Das Ergebnis sähe dann etwa in einer Access-Datenbank wie in Bild 1 aus.

In der objektorierten Programmierung sieht es ähnlich aus: Auch hier könnten Sie natürlich eine Klasse entwerfen, die sowohl die Basisdaten einer Person aufnehmen als auch die speziellen Daten für Mitarbeiter oder Kunden. Allerdings wird es dann schwer, bei der Programmierung immer nur genau die für den jeweiligen Typ gültigen Eigenschaften und Methoden zu nutzen, wenn immer alle Member etwa per IntelliSense angeboten werden.

Eine Alternative wäre es, jeweils eine Klasse für die Kunden und eine für die Mitarbeiter zu erstellen. Dabei würden beide jedoch eine Reihe identischer Eigenschaften enthalten wie zum Beispiel **Vorname** und **Nachname**. Und vielleicht gibt es auch noch Methoden, die Sie in beiden Klassen implementieren würden – wie etwa **GeburtstagskarteSchicken**. Diese

müssten Sie dann erstens doppelt anlegen und zweitens Änderungen immer direkt in zwei Klassen durchführen.

Und hier kommt die Vererbung ins Spiel – auch die Ableitung von Klassen genannt: Sie erstellen dabei genau wie bei der Basistabelle für die Personendaten erst einmal eine Basisklasse etwa namens **Person**. Diese stellt Eigenschaften bereit, mit denen Sie Daten wie **Vorname** oder **Nachname** einstellen können.

Wenn Sie nun einen Kunden anlegen möchten, benötigen Sie eine weitere Klasse: nämlich eine solche, die von der Klasse Person erbt beziehungsweise davon abgeleitet ist und die zusätzlich die für einen Kunden spezifischen Eigenschaften und Methoden hinzufügt. Diese wollen wir schlicht **Kunde** nennen. Das Gleiche erledigen wir für die spezielle Ausprägung einer Person namens Mitarbeiter - wir legen eine neue Klasse namens **Mitarbeiter** an, die von der Klasse **Person** erbt und ihre eigenen Eigenschaften und Methoden beisteuert.

Vererbung programmieren

Wir wollen uns an den Beispielfeldern der Tabellen aus dem obigen Datenbankbeispiel orientieren und entsprechende Klassen erstellen.

Wir benötigen also zunächst die Klasse **Person**, die wir wie folgt in der Datei **Program.cs** anlegen:

```
class Person {
    public string Vorname { get; set; }
    public string Nachname { get; set; }
}
```

Das ist die Kurzfassung, wenn Sie die Eigenschaften **Vorname** und **Nachname** nur lesen und schreiben möchten, aber kein weiterer Zugriff auf die Werte erforderlich ist. Wenn Sie beispielsweise in einer Eigenschaft namens Bezeichnung den **Vornamen** und den **Nachnamen** zusammengesetzt zurückgeben wollen, müssen Sie folgende Klassendefinition verwenden:

```
class Person {
    private string vorname;
    private string nachname;
    public string Vorname {
        get {
            return vorname;
        }
        set {
            vorname = value;
        }
    }
    public string Nachname {
        get {
            return nachname;
        }
        set {
            nachname = value;
        }
    }
    public string Bezeichnung {
        get {
            return vorname + " " + nachname;
        }
    }
}
```

Hier gibt es also interne Variablen namens **vorname** und **nachname**, welche die über die Eigenschaften **Vorname** und **Nachname** übergebenen Daten speichern. Für beide Eigenschaften legen wir jeweils einen **get**- und einen **set**-Block an, der übergebene Werte in der entsprechenden Variablen speichert oder diesen wieder herausgibt. Die dritte Eigenschaft heißt **Bezeichnung** und soll eine aus den Inhalten von **vorname** und **nachname** zusammengesetzte Zeichenkette zurückliefern.

Die Klasse nutzen Sie beispielsweise wie folgt:

```
public void KlasseOhneVererbung() {
    Person person = new Person();
    person.Vorname = "André";
```

```
    person.Nachname = "Minhorst";
    Console.WriteLine("Bezeichnung: {0}", person.Bezeichnung);
    Console.ReadLine();
}
```

Die Klasse Kunde

Nun erstellen wir die Klasse **Kunde**, welche von der Klasse **Person** erbt. Mit dieser Klasse hat es kaum etwas Besonderes auf sich – sie enthält vier private Variablen, die über die entsprechenden **get**-/**set**-Eigenschaften von außen lesend und schreibend zugreifbar sind. Der feine Unterschied zu einer einfachen Klasse ist jedoch, dass hinter dem Klassennamen ein Doppelpunkt gefolgt vom Namen der Klasse **Person** steht:

```
class Kunde : Person {
    private string firma;
    private string ustIdNr;
    private string benutzername;
    private string kennwort;
    public string Firma {
        get { return firma; }
        set { firma = value; }
    }
    public string UstIdNr {
        get { return ustIdNr; }
        set { ustIdNr = value; }
    }
    public string Benutzername {
        get { return benutzername; }
        set { benutzername = value; }
    }
    public string Kennwort {
        get { return kennwort; }
        set { kennwort = value; }
    }
}
```

Dadurch, dass die Klasse **Kunde** von der Klasse **Person** erbt, finden Sie in der Liste der Eigenschaften und Methoden der Klasse **Kunden** beim Zugriff über IntelliSense nun nicht mehr

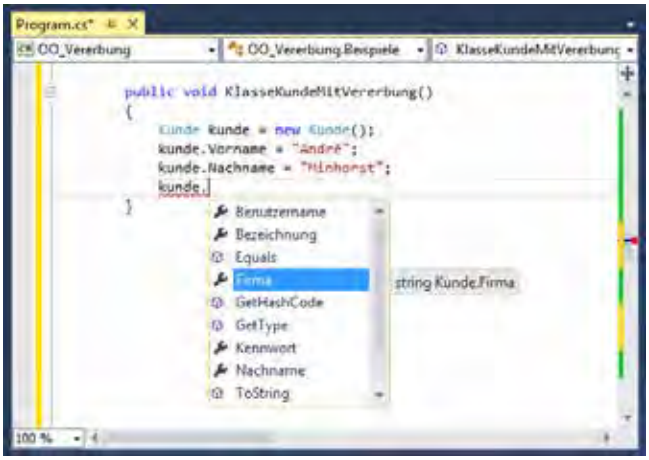


Bild 2: Auswahl der Eigenschaften einer erbenden Klasse

nur die Member der Klasse **Kunden**, sondern auch noch die der Klasse **Person** – also beispielsweise auch **Vorname** oder **Nachname** (siehe Bild 2).

Der Zugriff erfolgt dann durch Deklarieren und Initialisieren des Objekts **kunde** auf Basis der Klasse **Kunde**. Hier ist es dann unerheblich, dass sich hinter der Klasse eigentlich zwei Klassen verbergen:

```
public void KlasseKundeMitVererbung() {
    Kunde kunde = new Kunde();
    kunde.Vorname = "André";
    kunde.Nachname = "Minhorst";
    kunde.Benutzername = "aminhorst";
    kunde.Firma = "André Minhorst Verlag";
    kunde.Kennwort = "geheim";
    kunde.UstIdNr = "DE123456789";
}
```

Die Klasse Mitarbeiter

Auf die gleiche Weise erstellen wir die Klasse **Mitarbeiter**, die ebenfalls von der Klasse **Person** erbt:

```
class Mitarbeiter : Person {
    private string abteilung;
    private string funktion;
    public string Abteilung {
        get { return abteilung; }
```

Experimentieren mit der Konsole

Wenn Sie in die Programmierung mit C# einsteigen, werden Sie – genau wie wir in unseren Beiträgen – eine Menge ausprobieren wollen. Als Umsteiger von VBA haben Sie es da schwer: Unter VBA haben Sie zum Ausprobieren von Code einfach eine neue Prozedur in einem Standardmodul angelegt und diese dann mit F5 gestartet. In Visual Studio verwenden Sie zum Testen von Code zum Beispiel eine Konsolenanwendung. Verschiedene Prozeduren schreiben und mal eben aufrufen wie im VBA-Editor gelingt dort allerdings nicht. Wir zeigen Ihnen, wie Sie dennoch einigermaßen komfortabel experimentieren können.

Eine Konsolenanwendung für C# unter Visual Studio reicht leicht aus, um sich in die Sprache einzuarbeiten. Das Experimentieren damit ist allerdings nicht ganz so einfach wie etwa im VBA-Editor, wo man fernab von Klassenmodulen auch noch Standardmodule zum Programmieren einfacher und sofort aufrufbarer Prozeduren zur Verfügung hatte. Einfach die Prozedur eintippen, die Einfügemarke auf der gewünschten Prozedur platzieren, auf **F5** drücken – los geht es.

Unter C# wird aus den vorhandenen Klassen zunächst eine ausführbare **.exe**-Datei erstellt, die dann aufgerufen wird. Diese **.exe**-Datei hat genau einen Einstiegspunkt, nämlich die Methode **Main** in der Klasse **Program.cs**. Wenn Sie also irgendwelche Codezeilen testen wollen, führt kein Weg daran vorbei, die Konsolenanwendung zu kompilieren, die **.exe**-Datei zu erzeugen und diese zu starten. Gnädigerweise bietet Visual Studio diese Schritte alle über **F5** beziehungsweise den Menüeintrag **Debuggen|Debugging starten** an.

Nun wollen wir aber vielleicht nicht nur eine Prozedur oder, um im C#-Sprachgebrauch zu bleiben, eine Methode verwenden, um unsere vielen Ideen auszuprobieren (und auch in unserer Beispielklasse zu verewigen).

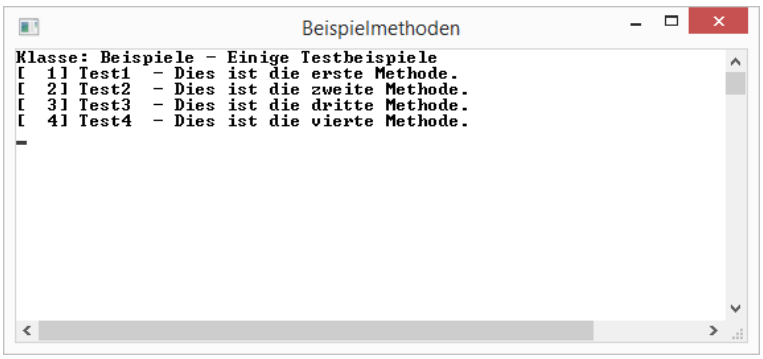


Bild 1: Anzeige der Beispielmethoden einer Klasse

Wie also können wir unsere Beispiele in verschiedene Methoden auslagern und diese dennoch komfortabel testen? Wir verwenden die Einstiegsmethode **Main** genau für diesen Zweck. Dazu fügen wir ihr ein paar Zeilen Code hinzu, welche die verschiedenen Beispiele nach dem Start der Beispielanwendung in der Konsole auflisten – also beispielsweise wie in Bild 1.

Wenn der Benutzer nun eine der in den eckigen Klammern angegebene Zahlen eingibt und die Eingabetaste betätigt, soll die entsprechende Funktion ausgeführt werden. Dies programmieren wir im einfachsten Fall so wie in Listing 1.

Diese Methode stellt zunächst einige Eigenschaften des Konsolen-Fensters ein – zugegebenermaßen hauptsächlich, damit die Screenshots der Artikel in **DATENBANKENTWICKLER** nicht so dunkel daherkommen. Danach durchläuft die Methode die folgenden Anweisungen in einer **do**-Schleife – und zwar so lange, bis die **String**-Variable **methodeNummer** keine leere Zeichenfolge enthält.

Diese Variable nimmt die Benutzereingabe nach der Anzeige der verfügbaren Beispiel-Methoden entgegen, also die Nummer der auszuführenden Methode. Diese Nummern werden samt Methodenname und einer kurzen Beschreibung durch die ersten Anweisungen innerhalb der **do**-Schleife

```
static void Main(string[] args) {
    Console.Title = "Beispielmethoden";
    Console.BackgroundColor = ConsoleColor.White;
    Console.ForegroundColor = ConsoleColor.Black;
    Console.Clear();
    String methodeNummer = "";
    do {
        Console.WriteLine("Klasse: Beispiel");
        Console.WriteLine("[ 1] Test1 - Dies ist die erste Methode.");
        Console.WriteLine("[ 2] Test2 - Dies ist die zweite Methode.");
        Console.WriteLine("[ 3] Test3 - Dies ist die dritte Methode.");
        Console.WriteLine("[ 4] Test4 - Dies ist die vierte Methode.");
        methodeNummer = Console.ReadLine();
        switch (methodeNummer) {
            case "1":
                Beispiele.Test1();
                break;
            case "2":
                Beispiele.Test2();
                break;
            case "3":
                Beispiele.Test3();
                break;
            case "4":
                Beispiele.Test4();
                break;
            default:
                break;
        }
    } while (methodeNummer != "");
}
```

Listing 1: Anzeige der Testfunktionen und deren Aufruf per Eingabe der entsprechenden Zahl

ausgegeben. Nach der Eingabe etwa der Zahl **1** für die erste Methode prüft die Prozedur in einer **switch**-Bedingung den Wert von **methodeNummer**. Im Falle der Zahl **1** landet diese im ersten **case**-Zweig und ruft die Methode **Test1()** der Klasse **Beispiele** auf.

Für die Beispiele legen Sie also dementsprechend eine eigene Klasse namens **Beispiele** an und fügen dort die gewünschten Methoden wie in Listing 2 ein. Die Methoden müssen Sie mit den Schlüsselwörtern **public static** versehen, damit Sie diese ohne Instanzieren eines Objekts auf Basis der Klasse aufrufen können. Die Methoden enthalten lediglich eine einfache **Console.WriteLine**-Anweisung, die Letzte gibt einen **integer**-Wert zurück.

```
public class Beispiele {
        public static void Test1() {
            Console.WriteLine("Test 1");
        }
        public static void Test2() {
            Console.WriteLine("Test 2");
        }
        public static void Test3() {
            Console.WriteLine("Test 3");
        }
        public static int Test4() {
            return 3;
        }
    }
```

Listing 2: Beispiel für eine Klasse mit Testprozeduren

Was ist zu tun? Um neue Beispielmethoden schnell auszuprobieren und die vorhandenen Beispiele ebenfalls im schnellen Zugriff zu haben, müssen Sie also neben einer neuen Methode zwei Dinge erledigen: das Hinzufügen einer **Console.WriteLine**-Anweisung zu Beginn der **do**-Schleife der Methode **Main** sowie eines neuen Zweiges in der **switch**-Bedingung.

Geht es auch weniger aufwendig?

Nachdem ich auf diese Weise einige Beispielmethoden erstellt und über die **Main**-Methode startbereit gemacht habe, stellte ich fest, dass dies doch immer einige Schritte sind. Am schönsten wäre es doch, wenn man einfach nur die neue Beispielmethode anlegen und diese dann einfach in der Liste der Beispielmethoden in der Konsole

vorfinden würde. Das heißt also, dass wir irgendwie dynamisch auf die Methoden zugreifen müssen. Unter VBA hätte man mit den Methoden der Objektbibliothek **Microsoft Visual Basic Extensibility 5.3 Object Library** auf den Quellcode der Beispielmethoden zuzugreifen und so per VBA den Code zu generieren, der für den Aufruf der Methoden nötig ist. Nun wissen wir ja bereits, dass man einfache Routinen in Standardmodulen unter VBA einfach so aufrufen kann, weshalb dieser Aufwand gar nicht nötig ist.

Aber wie können wir uns unter C# behelfen? Die Lösung ist eine Technik namens **Reflection**.

Zugriff auf den Code mit Reflection

Die **Reflection**-Klasse von C# machen Sie mit der entsprechenden **using**-Anweisung verfügbar:

```
using System.Reflection;
```

Danach ändern wir die Methode **Main**, die zuvor noch die manuell erstellte Liste der verfügbaren Beispielmethoden enthielt und diese aufgerufen hat, durch Code, der automatisch zur Laufzeit die Beispielklasse nach Methoden durchsucht und diese in der Konsole ausgibt.

sonst erst später berücksichtigen würde und erhalten eine enorme Zeit- und Arbeitsersparnis.

Bevor wir hier einsteigen, schauen wir uns noch eine Erweiterung an, die den Spaß am Testen der Beispiele noch steigern wird: Natürlich kann man ausreichend sprechende Methodennamen für die Beispiele definieren. Aber manchmal ist es doch hilfreich, wenn man noch eine kurze Bemerkung hinzufügen kann – eben so, wie in der Abbildung weiter oben.

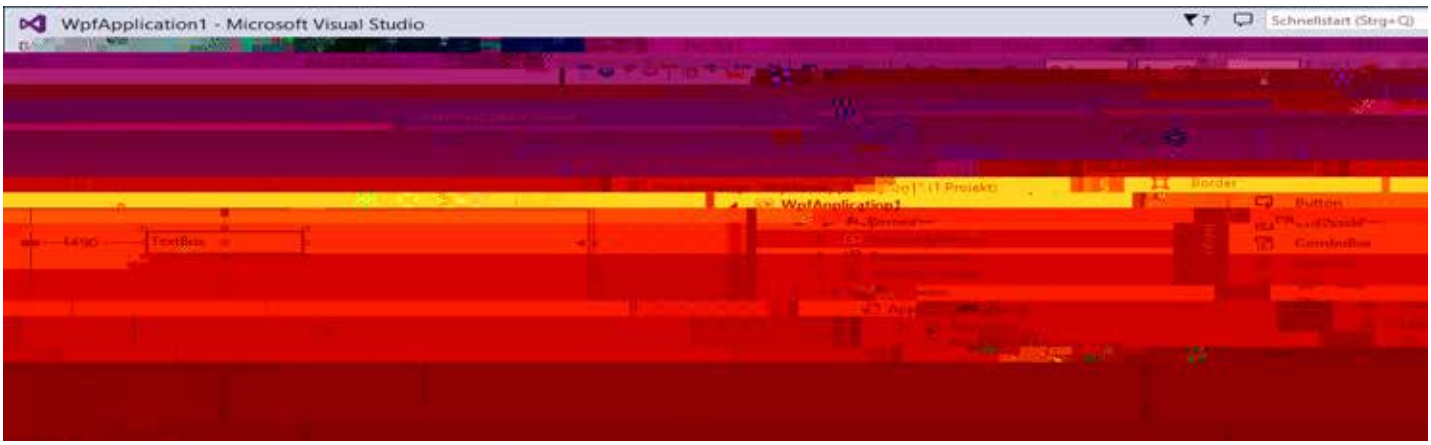
Wie aber wollen wir diese in den Methoden unterbringen, sodass diese auch eingelesen und in der Konsole ausgegeben werden kann? Kein Problem: Es gibt sogenannte benutzerdefinierte Attribute für die Elemente einer Klasse. Diese können Sie etwa wie in Listing 3 im Code der Klasse unterbringen, und zwar in dieser Form:

```
[Beschreibung("Einige Testbeispiele")]
public class Beispiele
{
    [Beschreibung("Dies ist die erste Methode.")]
    public static void Test1()
    {
        Console.WriteLine("Test 1");
    }
    [Beschreibung("Dies ist die zweite Methode.")]
    public static void Test2()
    {
```

AUSGABE 01/2016

DATENBANK ENTWICKLER

MAGAZIN FÜR DIE DATENBANKENTWICKLUNG MIT VISUAL STUDIO FÜR DESKTOP, WEB UND CO.



TOP-THEMEN:

C#-GRUNDLAGEN	Steuerelement-Ereignisse	SEITE 6
C#-GRUNDLAGEN	Dateidialoge	SEITE 18
ANWENDUNGEN	Anwendungskonfigurationsdateien	SEITE 29
DATENZUGRIFF	Datenzugriff mit ADO.NET, Teil 2	SEITE 40
VON ACCESS ZU WPF	Fenster mit einfachen Tabellendaten	SEITE 53

André Minhorst Verlag

Von VBA zu C#: Das Static-Schlüsselwort

Was bedeutet eigentlich das Schlüsselwort »static«, mit dem Sie sowohl Typen/Klassen also auch die Elemente einer Klasse wie Methoden oder Eigenschaften kennzeichnen können – und was ist bei der Erstellung und Nutzung von Typen und Elementen mit diesem Schlüsselwort zu beachten? Dies schauen wir uns im vorliegenden Artikel an und liefern einige Beispiele, welche die Regeln verdeutlichen.

Statische Klassen unter VBA

Unter VBA waren schon einige Verrenkungen nötig, wenn Sie auf die Methoden oder Eigenschaften einer Klasse zugreifen wollten, ohne diese Klasse selbst zu instanzieren – also beispielsweise so:

```
Dim objTest As clsTest
Set objTest = New clsTest
objTest.Beispielmethode
```

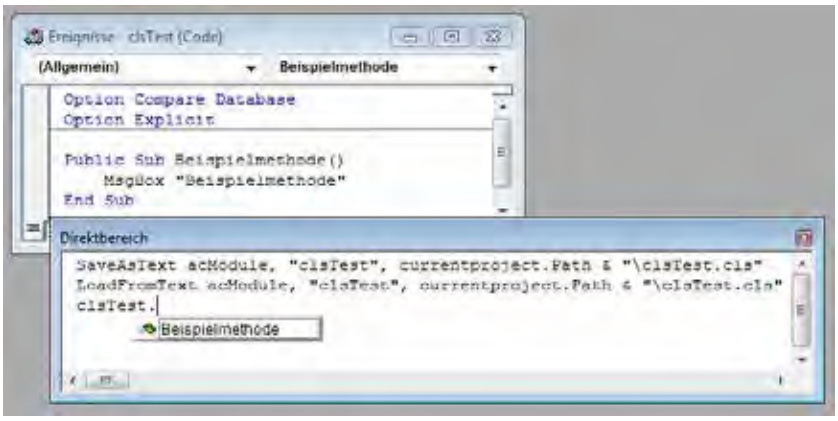


Bild 1: »Statische« Klassen unter VBA

Mit ein paar Tricks konnten Sie eine Klasse mit einer zusätzlichen, im VBA-Editor nicht sichtbaren Eigenschaft ausstatten, die es ermöglichte, direkt auf ihre Methoden zuzugreifen. Das sah dann schlicht so aus:

```
clsTest.Beispielmethode
```

Der Trick ist, die Klasse per **SaveAsText** zu exportieren und dann den Wert des Attributs **VB_PredeclaredId** in einem Texteditor auf **True** einzustellen:

```
Attribute VB_GlobalNameSpace = False
Attribute VB_Creatable = False
Attribute VB_PredeclaredId = True
Attribute VB_Exposed = False
Option Compare Database
Option Explicit

Public Sub Beispielmethode()
    MsgBox "Beispielmethode"
End Sub
```

Danach konnten Sie in VBA einfach den Namen der Klasse gefolgt von einem Punkt angeben, um per IntelliSense die Member der Klasse einzublenden (siehe Bild 1). Sie konnten die Klasse aber weiterhin instanzieren und wie eine herkömmliche Klasse per Objektverweis nutzen.

Zum Glück waren die beschriebenen Schritte nur notwending, wenn Sie die gewünschte Funktion tatsächlich im Kontext einer Klasse benötigten – also etwa, um Eigenschaften einzugeben und dann eine oder mehrere Methoden aufzurufen. Für den direkten Aufruf einer Methode (beziehungsweise Prozedur, wie es unter VBA heißt) gibt es ja zum Glück das gute, alte Standardmodul.

Statische Klassen unter C#

Unter C# ist dieses Feature standardmäßig vorgesehen, und zwar in verschiedenen Stufen. Dabei hilft Ihnen das Schlüsselwort **static**, das Sie dem Namen einer Klasse oder auch einer Methode, Variable oder anderen Membern voranstellen können – allerdings nicht in beliebigen Kombinationen.

In einer Klasse, die nicht mit dem Schlüsselwort **static** versehen ist, können Sie etwa eine Methode mit diesem Schlüsselwort ausstatten. Dies ist immer der Fall, wenn Sie etwa eine Konsolenanwendung erstellen – hier wird standardmäßig eine statische Methode namens **Main** angegeben, welche beim Start ausgelöst wird:

```
class Program {
    static void Main(string[] args) {
        ...
    }
}
```

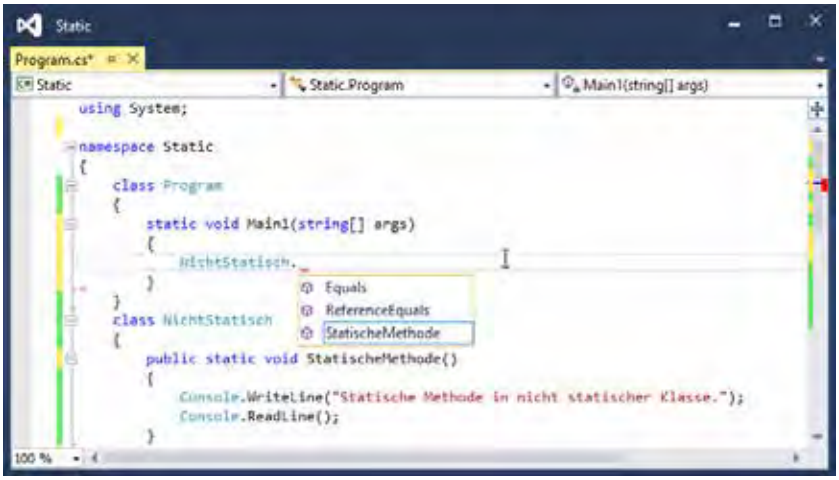


Bild 2: Statische Methode einer nicht statischen Klasse unter C#

Wollen Sie eine eigene Methode nutzen, ohne die Klasse als Objekt zu instanzieren, müssen Sie diese schlicht mit dem Schlüsselwort **static** auszeichnen. Dies erledigen Sie in beliebigen Klassen wie folgt:

```
class NichtStatisch {
    public static void StatischeMethode() {
        Console.WriteLine("Statische Methode in nicht
            statischer Klasse.");
        Console.ReadLine();
    }
}
```

Diese Methode rufen Sie nun etwa von der **Main**-Methode des Projekts wie folgt auf:

```
static void Main1(string[] args) {
    NichtStatisch.StatischeMethode();
}
```

Beachten Sie, dass Sie hier den Namen der Klasse beziehungsweise des Typs angeben, in dem sich die Methode befindet, und dann nach Eingabe eines Punktes direkt die statische Klasse per IntelliSense auswählen können (siehe Bild 2). Wenn Sie eine Methode hingegen nicht als statisch deklarieren, sieht diese wie folgt aus:

```
class NichtStatisch {
    public void NichtStatischeMethode() {
        Console.WriteLine("Nicht statische Methode in
            nicht statischer Klasse.");
        Console.ReadLine();
    }
}
```

Um diese aufzurufen, müssen Sie die Klasse zunächst instanzieren und in einer Objektvariablen speichern:

```
NichtStatisch nichtStatisch = new NichtStatisch();
nichtStatisch.NichtStatischeMethode();
```

Eine nicht als statisch deklarierte Klasse mit statisch deklarierten Methoden verhält sich aber dennoch nicht so wie die eingangs beschriebene VBA-Klasse – die statischen Methoden stehen zwar direkt über den Typbezeichner zur Verfügung und die nicht statischen Methoden über die Instanz.

Aber die statischen Member können ausschließlich über den Typ und nicht über eine Instanzvariable genutzt werden.

Statische Klassen/Typen

Es gibt noch eine weitere Variante, nämlich die, bei der Sie die komplette Klasse als statisch deklarieren – etwa so:

Von VBA zu C#: Steuerelement-Ereignisse

Wer unter VBA Ereignisprozeduren für Formulare, Berichte oder Steuerelemente erstellen wollte, hatte es leicht: Einfach in der Ereigniseigenschaft den Wert [Ereignisprozedur] auswählen, auf die Schaltfläche mit den drei Punkten klicken und schon konnte man die vorgefertigte Prozedur mit Anweisungen füllen. In manchen Fällen war das Implementieren von Ereignissen schon komplizierter, aber immer noch schnell machbar. Unter C# bekommt man dies auch meist mit wenigen Klicks hin, aber die notwendigen Handgriffe unterscheiden sich doch deutlich von denen unter VBA. Dieser Artikel zeigt, wie Sie gängige Varianten von Ereignisprozeduren anlegen.

Unter einem Ereignis verstehen wir dabei etwas, das durch eine bestimmte Aktion ausgelöst wird. Unter Access steht dies meist in Zusammenhang mit einem Formular oder einem Steuerelement, zum Beispiel das Öffnen oder Schließen des Formulars, das Anklicken einer Schaltfläche oder das Ändern des Inhalts eines Textfeldes.

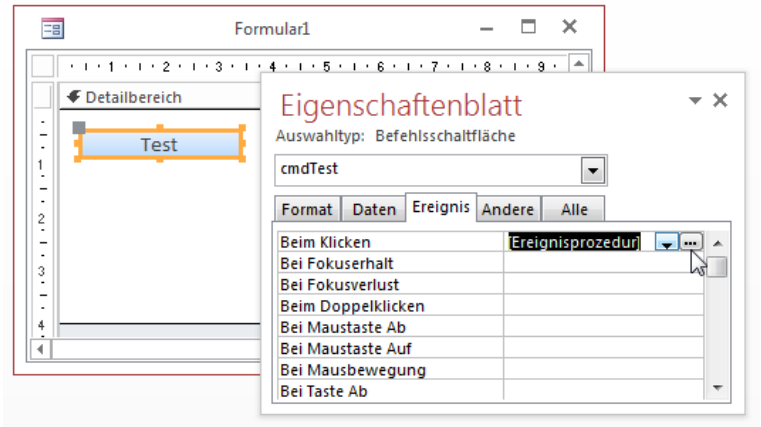


Bild 1: Anlegen einer Ereignisprozedur

Für die Implementierung einer Prozedur, die beim Eintreten des Ereignisses ausgelöst wird, sind dann zwei Schritte nötig: Erstens haben wir die entsprechende Ereigniseigenschaft aus dem Eigenschaftsfenster herausgesucht und dort den Wert **[Ereignisprozedur]** ausgewählt (siehe Bild 1).

Zweitens haben wir durch einen Klick auf die Schaltfläche mit den drei Punkten neben dem Eigenschaftswert die leere Prozedur angelegt, die dann wie in Bild 2 aussieht – die Ereignisprozedur. Wenn wir hier eine Anweisung wie **MsgBox "Test"** einfügen, in die Formularansicht wechseln und die Schaltfläche betätigen, wurde die Prozedur wie gewünscht ausgewählt.

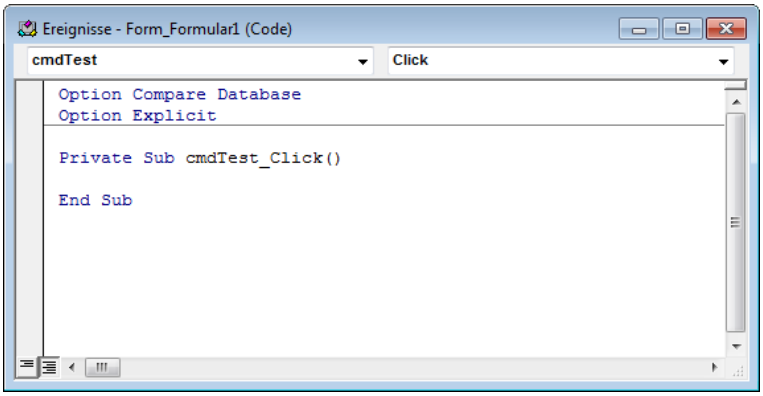


Bild 2: Eine neue Ereignisprozedur unter VBA/Access

Hier gibt es eine implizite Vereinbarung: Diese besagt, dass die Definition der Ereignisprozedur bestimmten Regeln entsprechen muss. Die erste bezieht sich auf den Namen, der aus dem Steuerelementnamen (oder Formular-/Berichtsnamen), einem Unterstrich sowie der englischen Version der Ereignisbezeichnung bestehen muss, hier beispielsweise **cmdText_Click**. Da es jedes Ereignis für jedes Objekt/Steuerelement nur einmal gibt, funktioniert dies sehr gut und

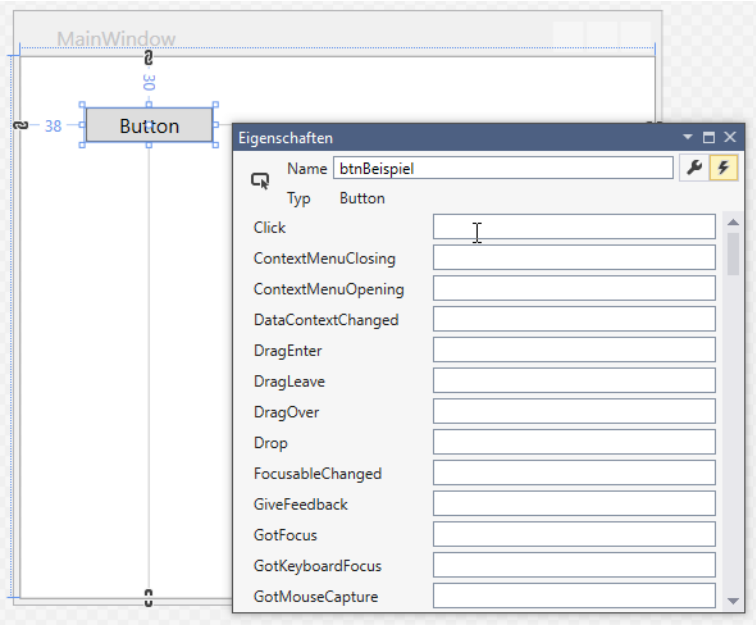


Bild 3: Anlegen einer Ereignisprozedur unter WPF

man braucht für die Ereigniseigenschaft nur den Wert **[Ereignisprozedur]** anzugeben statt etwa des Prozedurnamens.

Andersherum können Sie Ereignisprozeduren auch über den VBA-Editor anlegen. Dazu wählen Sie im linken Kombinationsfeld den Namen des Objekts aus, das Sie mit dem Ereignis ausstatten wollen, und im linken das entsprechende Ereignis. Der VBA-Editor legt dann sowohl die leere Ereignisprozedur an als auch den Eintrag in der entsprechenden Ereigniseigenschaft.

Fenster- und Steuerelementereignisse unter WPF

Unter WPF läuft es ähnlich ab. Für die folgenden Beispiele verwenden wir ein C#/WPF-Projekt namens **Ereignisse**. Um für eine Schaltfläche eine Ereignisprozedur zu implementieren, die beim Anklicken der Schaltfläche ausgelöst wird, erstellen Sie zunächst eine neue Schaltfläche, in diesem Fall direkt im Hauptfenster der Anwendung namens **MainWindow**.

Wir benennen die Schaltfläche um, indem wir das Attribut **name** des **button**-Objekts im XAML-Bereich mit dem gewünschten Namen versehen:

```
<Button x:Name="btnBeispiel" .../>
```

Neue Ereignisprozedur per Eigenschaftsfenster

Danach gibt es eine einfache Möglichkeit, wie unter Access/VBA gleichzeitig die Ereigniseigenschaft zu füllen als auch die Ereignisprozedur anzulegen. Dazu wechseln Sie im Eigenschaftsfenster, das standardmäßig die allgemeinen Eigenschaften für das ausgewählte Objekt anzeigt, auf den Bereich für die Ereignishandler. Dazu ist lediglich ein Klick auf die Schaltfläche mit dem Blitz-Symbol nötig (siehe Bild 3).

Hier klicken Sie dann einfach doppelt in die für uns interessante Eigenschaft, nämlich die mit der Bezeichnung **Click**. Das ist im Vergleich zu Access etwas weniger intuitiv, denn dort lässt sich der Wert **[Ereignisprozedur]** ja per Kombinationsfeld auswählen und die Prozedur per Klick auf die Schaltfläche mit den drei Punkten anlegen. Das Ergebnis ist, dass die Eigenschaft mit dem Wert **btnBeispiel_Click** gefüllt wird und eine neue, leere Methode in der Klasse **MainWindow** mit dem Code für das WPF-Fenster erscheint:

```
private void btnBeispiel_Click(object sender,
    RoutedEventArgs e) {
}
```

Die Methode wird bereits mit den benötigten Parametern ausgestattet (zu diesen kommen wir gleich). Der Unterschied zu Access/VBA ist nun, dass wir für die Ereigniseigenschaft nicht pauschal einen Wert wie **[Ereignisprozedur]** eintragen und der Name der durch das Ereignis ausgelösten Methode bestimmten Vorgaben entsprechen muss. Stattdessen könnten Sie auch jeden anderen Wert für die Eigenschaft **Click** eintragen, solange die auszulösende Methode entsprechend benannt wird. Aber wo speichert Visual Studio nun die Eigenschaft und den angegebenen Wert tatsächlich? Dies geschieht direkt in der XAML-Definition des Fensters, in diesem Fall mit dem Attribut **Click**:

Von VBA zu C#: Objekt-Ereignisse

Nicht nur Fenster und Steuerelemente, sondern auch Objekte, die nicht Bestandteil der Benutzeroberfläche sind, können Ereignisse auslösen. Schließlich können Sie auch selbst Klassen programmieren und diese mit Ereignishandlern versehen. Wie Sie unter C# mit diesen Möglichkeiten umgehen, zeigt dieser Artikel. Dabei schauen wir uns zunächst an, wie Sie Ereignisse für Objekte auf Basis etwa der OpenFileDialog-Klasse implementieren und erstellen dann eine benutzerdefinierte Klasse mit einem Ereignishandler.

Ereignisse von Klassen implementieren

Neben Formularen und Steuerelementen bieten unter Access auch verschiedene Klassen beziehungsweise Objekte Ereignisse an, die Sie selbst implementieren können. Als Beispiel sei etwa die Klasse **Application** anderer Office-Anwendungen wie Outlook,Word oder Excel genannt. Sie deklarieren dabei eine entsprechende Objektvariable wie etwa **objWord** mit dem Typ **Word.Application** und instanzieren Word dann mit der **New**-Methode. Wenn Sie die Deklaration mit dem Schlüsselwort **WithEvents** durchführen, können Sie anschließend nicht nur Methoden oder Eigenschaften von Word aufrufen, sondern auch noch die Ereignisse von Word durch entsprechende Ereignisprozeduren implementieren – beispielsweise, wenn ein Dokument geöffnet oder geschlossen wird:

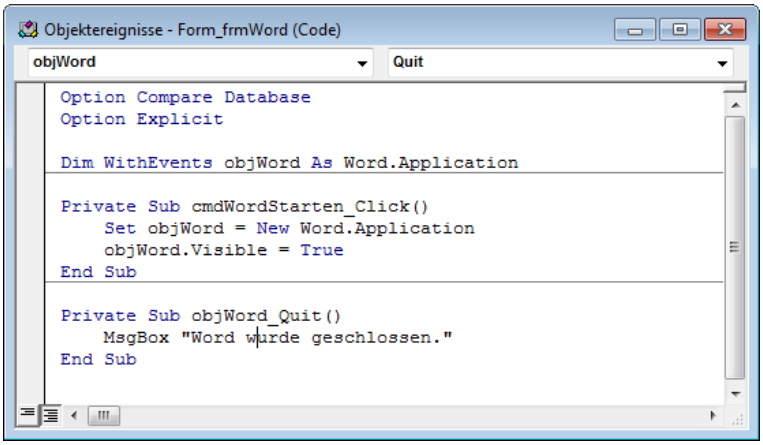


Bild 1: Implementieren eines Ereignisses eines Objekts

```
Dim WithEvents objWord As Word.Application
Set objWord = New Word.Application
```

Dazu wählen Sie dann im linken Kombinationsfeld des VBA-Fensters, in der die Variable **objWord** definiert ist, den Eintrag **objWord** aus und können dann mit dem rechten Kombinationsfeld die zur Verfügung stehenden Ereignisse selektieren – beispielsweise das **Quit**-Ereignis. Im Beispiel aus Bild 1 haben wir etwa eine Word-Instanz erstellt und unter **objWord** gespeichert. Die für das **Quit**-Ereignis erstellte Ereignisprozedur haben wir mit einer Anweisung gefüllt, die ein Meldungsfenster liefert, wenn die Word-Instanz wieder geschlossen wird.

Ereignis unter C# implementieren

Unter C# läuft dies etwas anders, aber nicht viel. Im folgenden Beispiel wollen wir die **Timer**-Klasse verwenden (der Zugriff auf Word ist von C#-Projekten aus etwas aufwändiger – dies schauen wir uns in einem späteren Beitrag an ...). Die **Timer**-Klasse erlaubt es, ein Zeitgeber-Intervall festzulegen, nach dem das Ereignis **Elapsed** ausgelöst wird.

Die Objektvariable für das Objekt, das ein Ereignis auslösen soll, deklarieren wir unter C# nicht mit einem speziellen Schlüsselwort wie **WithEvents** unter VBA, sondern einfach als private, statische Variable:

```
private static Timer timer;
```

In der gleichen Klasse legen wir unsere Methode an, welche das Objekt auf Basis der Klasse **Timer** instanziert, stellen das Zeitgeber-Intervall ein und starten den Zeitgeber. Außerdem,

und das ist der wichtigste Punkt, legen wir fest, dass beim Eintreten des Ereignisses **Elapsed** der **Timer**-Klasse die Methode **Timer_Elapsed** ausgeführt werden soll. Dies erreichen wir durch die folgende Zeile:

```
timer.Elapsed += Timer_Elapsed;
```

Die komplette Prozedur zum Erstellen des **Timer**-Objekts und zum Starten des Zeitgebers sieht nun wie folgt aus:

```
public static void TimerEreignis() {
    Console.WriteLine("Start Timer. Intervall: eine Sekunde.");
    timer = new Timer();
    timer.Elapsed += Timer_Elapsed;
    timer.Interval = 1000;
    timer.Start();
}
```

Nun benötigen wir noch die Methode, die beim Eintreten des **Elapsed**-Ereignisses ausgelöst wird – unter VBA haben wir diese Ereignisprozedur genannt. Diese erhält die ähnliche Signatur wie auch die Ereignisprozeduren etwa von Steuerelementen, also mit zwei Parametern namens **sender** und **e**:

```
private static void Timer_Elapsed(object sender,
        ElapsedEventArgs e) {
    timer.Stop();
    Console.WriteLine("Zeit abgelaufen.");
    Console.ReadLine();
}
```

e hat hier allerdings den Typ **ElapsedEventArgs**. Dieses Objekt liefert neben den Standardelementen mit der Eigenschaft **SignalTime** die Zeit, zu der das Ereignis ausgelöst wurde. Dieses nehmen wir noch zur Ausgabe hinzu:

```
Console.WriteLine("Zeit abgelaufen. {0}", e.SignalTime);
```

Eigene Ereignisse programmieren

Nun wollen wir eine Klasse mit einem eigenen Ereignis programmieren. Ausgangspunkt ist die folgende einfache Klasse namens **Person** mit einer einzigen Eigenschaft namens **Vorname** sowie den entsprechenden **get**- und **set**-Methoden:

```
public class Person {
    private string vorname;
    public string Vorname {
        get {
            return this.vorname;
        }
        set {
            this.vorname = value;
        }
    }
}
```

Die folgende Methode erstellt ein neues Objekt auf Basis dieser Klasse und füllt die Eigenschaft **Vorname** mit dem Wert **André**:

```
public static void KundeErstellen() {
    Person person = new Person();
    person.Vorname = "André";
    Console.WriteLine("Vorname: {0}", person.Vorname);
    Console.ReadLine();
}
```

Diese Klasse soll nun beim Ändern des Wertes der Eigenschaft Vorname ein Ereignis auslösen, das wir wiederum in der Klasse mit der Methode implementieren wollen. Dazu sind auf Seite der Klasse **Person** die folgenden Schritte nötig:

- Definieren eines Delegates, der festlegt, welche Parameter die zu implementierenden Ereigisprozeduren verwenden müssen (und die auch beim Auslösen gefüllt werden),
- Definieren eines Ereignisses mit dem Schlüsselwort event, dem Typ des Ereignisses (der dem zuvor definierten Delegate entspricht) und dem Namen des Ereignisses und
- die Definition einer Methode, die der Signatur des Delegates entspricht und das Ereignis auslöst, sofern dieses

abonniert wurde und die innerhalb der Klasse beim Eintreten des jeweiligen Ereignisses – hier beim Ändern des Wertes der Eigenschaft **Vorname** in der entsprechenden **Set**-Methode – aufgerufen wird.

```
public class Person {
    private string vorname;
    public string Vorname {
        get {
            return this.vorname;
        }
        set {
            this.vorname = value;
            BeiEigenschaftGeaendert(this, null);
        }
    }

    public delegate void EigenschaftGeaendertEventHandler(object sender, EventArgs e);

    public event EigenschaftGeaendertEventHandler EigenschaftGeaendert;

    protected void BeiEigenschaftGeaendert(object sender, EventArgs e) {
        if(EigenschaftGeaendert != null) {
            EigenschaftGeaendert(this, e);
        }
    }
}
```

Listing 1: Die Klasse **Person** mit einem Ereignis, das beim Ändern einer Eigenschaft feuert

Die Klasse **Person** wird nun also wie in Listing 1 erweitert. In der **get**/**set**-Methode kommt nur die Zeile hinzu, welche die Methode **BeiEigenschaftGeaendert** aufruft. Diese übergibt als ersten Parameter einen Verweis auf die Klasse selbst, als zweiten den Wert **null**. Später werden wir dies noch erweitern.

Das **delegate**-Objekt namens **EigenschaftGeaendertEventHandler** enthält die Signatur für die zu implementierende Ereignisprozedur. Die Namen aller Delegates, die als Typ eines **event**-Objekts herangezogen werden, sollen auf **EventHandler** enden.

Die Ereignis-Definition, also die Zeile mit dem Schlüsselwort **event**, enthält wie eine herkömmliche Variablendeklaration den Typ der Variablen (hier also **EigenschaftGeaendertEventHandler**) sowie den Namen des Events, hier **EigenschaftGeaendert**. Die Verwendung des **event**-Schlüsselworts bringt unter anderem den Vorteil, dass Sie in der definierenden Methode die Operatoren **+=** und **-=** verwenden können, um das Ereignis zu abonnieren oder zu entfernen.

Fehlt noch der Aufruf des Events. Diesen packen wir in eine eigene Methode namens **BeiEigenschaftGeaendert**, welche die Standardparameter für Ereignisse verwendet, also solche vom Typ **object** und **EventArgs**. Die Methode vergleicht den Wert des Events **EigenschaftGeaendert** mit dem Wert **null**. Dies dient der Prüfung, ob das Event überhaupt von der implementierenden Klasse abonniert wurde. »Abonniert« entspräche unter VBA dem Zuweisen des Wertes **[Event Procedure]** zu der jeweiligen Ereigniseigenschaft, also etwa so:

```
txt.AfterUpdate = "[Event Procedure]"
```

Unter C# gelingt dies, sofern Sie die Klasse **Person** wie oben beschrieben erweitert haben, durch einen kleinen Schritt: Dann stellen Objekte auf Basis der Klasse **Kunde** nämlich den Namen des **event**-Objekts, hier **EigenschaftGeaendert**, als Eigenschaft bereit. Diesem fügen Sie dann den Namen der Methode hinzu, mit der Sie das Ereignis implementieren wollen. Visual Studio unterstützt Sie dabei, indem es vorschlägt, die **Tabulator**-Taste zu drücken, um die Event-Eigenschaft zu füllen und das Grundgerüst des Eventhandlers anzulegen (siehe Bild 2).

Von VBA zu C#: Dateidialoge

Unter VBA musste man schon einigen Zusatzcode inklusive Api-Deklarationen zu seinem Projekt hinzufügen, um Dialoge etwa zum Auswählen einer zu öffnenden Datei, eines Verzeichnisses oder zur Angabe eines Dateinamens zum Speichern anzuzeigen. Unter .NET gibt es dazu natürlich eine vorgefertigte Klasse, die alle notwendigen Funktionen liefert. Der vorliegende Artikel zeigt, wie Sie diese Dialoge anzeigen und die damit ermittelten Daten nutzen können.

Die folgenden Beispiele verwenden eine WPF-Anwendung unter C#.

Dateien auswählen per Klasse

Unter VBA gibt es verschiedene Varianten, um beispielsweise einen **Datei öffnen**-Dialog anzuzeigen. Per API, über die versteckte Wizhook-Klasse oder auch mit der entsprechenden Klasse der Office-Bibliothek. Alle Lösungen hatten Vor- und Nachteile, aber wirklich perfekt war keine davon.

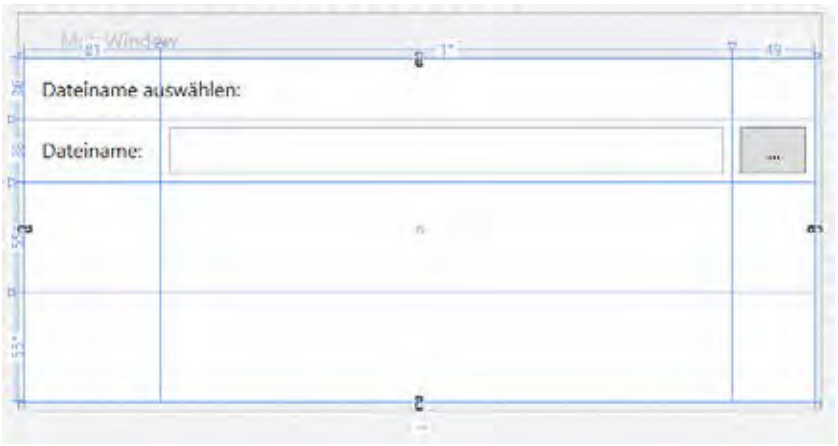


Bild 1: Entwurf des Fensters zum Auswählen einer Datei

Unter .NET und in unserem Fall unter C# gibt es eine Klasse namens **OpenFileDialog**. Diese ist Bestandteil des Namespaces **Microsoft.Win32**, den Sie der Klasse per **using**-Anweisung bekannt machen:

```
using Microsoft.Win32;
```

Danach können Sie bereits auf die **OpenFileDialog**-Klasse zugreifen. Dazu legen wir ein WPF-Formular an, das wie in Bild 1 aussieht. Das Textfeld heißt **txtDateiname**, die Schaltfläche **btnDateiAuswaehlen**. Für die Schaltfläche hinterlegen wir die folgende Ereignisprozedur:

```
private void btnDateiAuswaehlen_Click(object sender,
        RoutedEventArgs e) {
    OpenFileDialog openFileDialog = new OpenFileDialog();
    if (openFileDialog.ShowDialog()== true) {
        txtDateiname.Text = openFileDialog.FileName;
    }
}
```

Diese erstellt zunächst ein neues Objekt auf Basis der Klasse **OpenFileDialog**. Im folgenden Schritt rufen wir den **Datei öffnen**-Dialog mit der Funktion **ShowDialog** auf. Diese zeigt den **Datei öffnen**-Dialog an und erwartet die Eingabe des Benutzers (siehe Bild 2). Die Funktion liefert als Ergebnis einen Boolean-Wert zurück, den die **if**-Anweisung auswertet. Lautet das Ergebnis **true**, hat der Benutzer eine Datei ausgewählt und wir können diese über die Eigenschaft **FileName** des **OpenFileDialog**-Objekts auslesen und als Text des Textfeldes **txtDateiname** eintragen.

Den Wert **false** liefert die Funktion nur dann zurück, wenn der Benutzer den Dialog mit der **Schließen**-Schaltfläche oben rechts oder mit der **Abbrechen**-Schaltfläche unten rechts beendet.

Eigenschaften des OpenFileDialog-Objekts

Die Klasse besitzt natürlich einige Eigenschaften, mit denen Sie das Aussehen des Dialogs und auch seine Funktion

beeinflussen können. Hier ist eine Liste der wichtigsten Eigenschaften samt Kurzbeschreibung – weiter unten schauen wir uns Beispiele zu den wichtigsten Eigenschaften an:

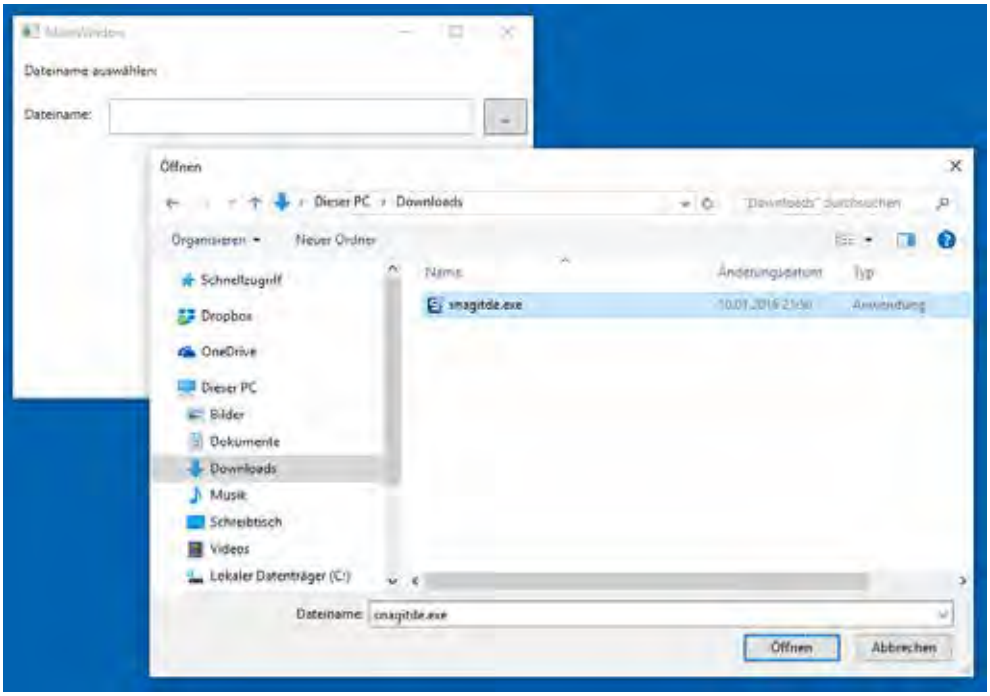


Bild 2: Ein einfacher **Datei öffnen**-Dialog

- **CheckFileExists**: Boolean-Wert, der angibt, ob das Vorhandensein der gewählten Datei geprüft werden soll. Der Wert **false** lässt auch die Angabe nicht vorhandener Dateien zu, der Wert **true** liefert eine Meldung, wenn eine nicht vorhandene Datei gewählt wird (Standardwert: **true**)

- **CustomPlaces**: Spezielle Ordner zusätzlich zu den Favoriten-Ordnern anzeigen

- **DereferenceLinks**: Legt fest, ob bei der Auswahl einer Dateiverknüpfung die Verknüpfung selbst zurückgegeben werden soll (in der Regel eine **.lnk**-Datei) oder der Pfad zu der verknüpften Datei.

- **FileName**: Liefert den Dateinamen der gewählten Datei zurück.

- **FileNames**: Liefert ein Array mit den Dateinamen der gewählten Dateien zurück. Sowohl **FileName** als auch **FileNames** funktionieren bei der Einfachauswahl als auch bei der Mehrfachauswahl (siehe Eigenschaft **MultiSelect**).

- **FileOk**: Dies ist ein Ereignis, das ausgelöst wird, wenn der Benutzer die Schaltfläche **Öffnen** des **Datei öffnen**-Dialogs betätigt hat.

- **Filter**: Erwartet einen Ausdruck, der die zu filternden Dateien angibt. Beispiel: **Access-Datenbanken (*.mdb, *.accdb)|*.mdb;*.accdb|Alle Dateien (*.*)|*.***

- **FilterIndex**: Legt fest, welcher **Filter**-Eintrag angezeigt werden soll (**1** für den ersten Eintrag, **2** für den zweiten und so weiter)

- **InitialDirectory**: Gibt das beim Einblenden des Dialogs anzuzeigende Verzeichnis an, entweder als Zeichenkette mit führendem @-Zeichen (**@"c:\"**) oder als Ausdruck, zum Beispiel **AppDomain.CurrentDomain.BaseDirectory** für das Verzeichnis der aktuellen Anwendung.

- **MultiSelect**: **Boolean**-Eigenschaft, die festlegt, ob nur eine (**false**) oder mehrere Dateien gleichzeitig ausgewählt werden können (**true**).

- **SafeFileName**: Liefert den Dateinamen ohne Verzeichnis.

- **SafeFileNames**: Liefert bei Mehrfachauswahl die Dateinahmen ohne Verzeichnis.

- **ShowDialog**: Zeigt den Dialog an und liefert den Wert **true** zurück, wenn der Benutzer den Dialog mit der **Öffnen**-Schaltfläche schließt.

- **Title**: Legt den Fenstertitel fest.

Mehrere Dateien ermitteln

Wenn Sie mit der **OpenFileDialog**-Klasse nicht nur eine, sondern gegebenenfalls auch einmal mehrere Dateien gleichzeitig ermitteln möchten, bedarf es nur weniger Änderungen (siehe Listing 1). Dazu stellen Sie zunächst einmal den Wert der Eigenschaft **MultiSelect** der Objektvariablen **openFileDialog** auf den Wert **true** ein. Danach zeigen Sie den Dialog wie gewohnt mit der Methode **ShowDialog** an.

```
private void btnDateienAuswaehlen_Click(object sender, RoutedEventArgs e) {
    OpenFileDialog openFileDialog = new OpenFileDialog();
    openFileDialog.InitialDirectory = AppDomain.CurrentDomain.BaseDirectory;
    openFileDialog.Multiselect = true;
    if (openFileDialog.ShowDialog() == true) {
        foreach(string filename in openFileDialog.FileNames) {
            lstDateinamen.Items.Add(filename);
        }
        txtDateiname.Text = openFileDialog.FileName;
    }
}
```

Listing 1: Einlesen mehrerer Dateinamen per **OpenFileDialog**

Hat der Benutzer die gewünschten Dateien ausgewählt und die **Öffnen**-Schaltfläche betätigt (siehe Bild 3), gehen wir allerdings etwas anders vor als bei einer einzigen Datei. In diesem Fall soll die Methode nämlich eine **foreach**-Schleife über alle **string**-Objekte der Auflistung **FileNames** des **OpenFileDialog**-Objekts durchlaufen. Innerhalb der Schleife fügt die Methode der Auflistung **Items** des Listenfeldes namens **lstDateinamen** einen neuen Eintrag mit der **Add**-Methode hinzu.

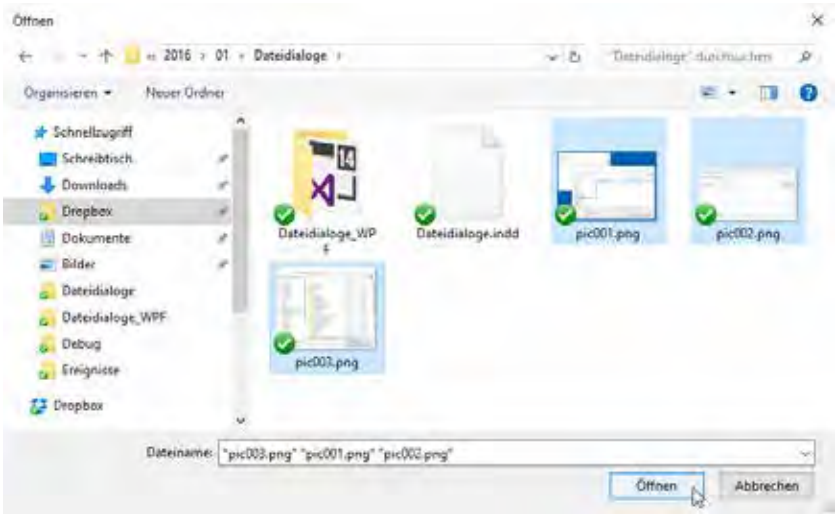


Bild 3: Markieren mehrerer Dateien gleichzeitig

Das Ergebnis sieht wie in Bild 4 aus. Der Methode haben wir noch eine weitere Anweisung hinzugefügt, welche den Wert der Eigenschaft **FileName** in das Textfeld **txtDateiname** einträgt. Damit zeigen wir, dass auch bei der Einstellung **MultiSelect = true** der erste Eintrag über die **FileName**-Eigenschaft ermittelt werden kann.

Textdatei einlesen

Wenn wir schon Dateien auswählen, können wir deren Inhalt auch direkt einmal in ein Textfeld einlesen – zumindest, wenn es sich um eine Textdatei handelt.

Dazu verwenden wir die Schaltfläche **btnDateiEinlesen**, welche die Methode aus Listing 2 auslöst. Wir

Bild 4: Anzeige der Dateien im Listenfeld

```
private void btnDateiEinlesen_Click(object sender, RoutedEventArgs e) {
    OpenFileDialog openFileDialog = new OpenFileDialog();
    StreamReader streamReader;
    openFileDialog.Filter = "Text-Dateien (*.txt)|*.txt|XML-Dateien (*.xml)|*.xml|Alle Dateien (*.*)|*.*";
    openFileDialog.FilterIndex = 1;
    if (openFileDialog.ShowDialog() == true) {
        string dateiname = openFileDialog.FileName;
        streamReader = new StreamReader(dateiname);
        this.txtDateiinhalt.Text = streamReader.ReadToEnd();
        this.txtDateiname.Text = dateiname;
    }
}
```

Listing 2: Einlesen einer Textdatei nach der Auswahl mit dem **OpenFileDialog**

wollen an dieser Stelle gleich ein Beispiel für einen Filter liefern. Der **Öffnen**-Dialog soll entweder nur Dateien mit der Dateiendung **.txt**, mit der Endung **.xml** oder alle Dateien anzeigen (***.***) – siehe Bild 5. Dazu geben wir der Eigenschaft **Filter** einen entsprechenden Ausdruck mit und stellen den zu Beginn anzuzeigenden Filtereintrag mit der Eigenschaft **FilterIndex** ein.

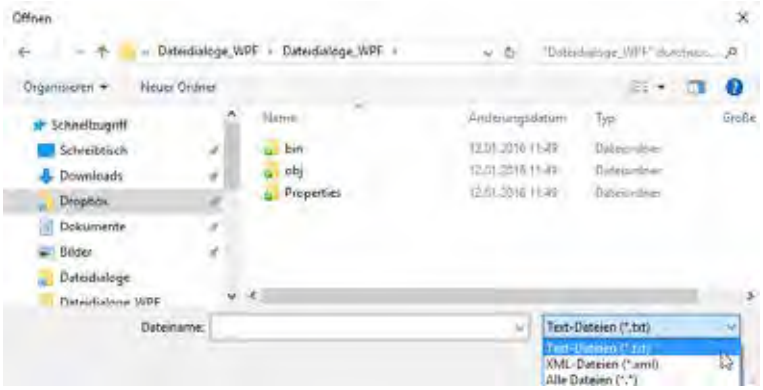


Bild 5: Einlesen von Text- oder XML-Dateien per Filter

Nach dem Anzeigen des **Öffnen**-Dialogs und der Auswahl der Datei speichert die Methode den Dateinamen in der Variablen **dateiname**. Dann erstellt sie ein neues Objekt des Typs **StreamReader** und übergibt dem Konstruktor den Dateinamen. Die Methode **ReadToEnd** ermittelt schließlich den Inhalt der Textdatei und trägt diesen in das Textfeld **txtDateiinhalt** ein. Der Dateiname landet schließlich noch im Textfeld **txtDateiname**. Das Ergebnis sieht dann wie in Bild 6 aus.

Bild 6: Das Beispielformular nach dem Einlesen einer XML-Datei

Ereignis nach Auswahl auslösen

Die Klasse **OpenFileDialog** bietet auch ein Ereignis an, das Sie implementieren können. Dazu legen wir eine neue Schaltfläche namens **btnEreignis-FileOk** an. Die dadurch ausgelöste Methode finden Sie in Listing 3. Diese zeigt wieder ein **OpenFileDialog**-Fenster an. Diesmal fügt sie jedoch der Eigenschaft **FileOk** den Namen der auszulösenden Methode hinzu, nämlich **OpenFileDialog1_FileOk**. Dazu verwenden wir den **+=**-Operator. Nach der Eingabe dieses Operators bietet Visual Studio an, per **Tabulator**-Taste automatisch den Wert **OpenFileDialog1_FileOk**

Delegates

Delegates sind Typen, denen Sie eine Methode zuweisen, die sie dann über das Delegate ausführen können. Ein Delegate legt dabei nur fest, welchen Rückgabewert und welche Parameter die zu verwendende Methode nutzt – Sie können dem Delegate zur Laufzeit dann beliebige Methoden zuweisen, die dieser Signatur entsprechen. Delegates benötigen Sie aber auch, wenn Sie Klassen mit benutzerdefinierten Ereignissen programmieren möchten. In diesem Fall nutzen Sie ein Delegate, um die beim Eintreten des Ereignisses auszuführende Methode festzulegen.

In diesem Artikel schauen wir uns zunächst an, wie ein Delegate funktioniert, und erläutern erst im Anschluss, welchen Nutzen dies hat. Angenommen, Sie haben zwei Methoden, welche die gleichen Parameter nutzen, aber verschiedene Dinge erledigen. Ein schönes Beispiel ist dabei das Addieren und das Multiplizieren zweier Zahlen. Die beiden Methoden würden dann etwa wie folgt aussehen:

```
public static int Summe(int a, int b) {
    return a + b;
}
public static int Produkt(int a, int b) {
    return a * b;
}
```

Wollen Sie nun eine dieser Methoden aufrufen, würden Sie dies beispielsweise wie folgt erledigen:

```
Console.WriteLine("Summe aus 2 und 3: {0}", Summe(2, 3));
```

Zum Multiplizieren rufen Sie einfach die Methode **Produkt** auf:

```
Console.WriteLine("Produkt aus 2 und 3: {0}", Produkt(2, 3));
```

Wenn Sie beim Erstellen des Projekts noch nicht wissen, welche der Methoden an einer bestimmten Stelle zum Einsatz kommen wird, fragen Sie beispielsweise den Benutzer danach und codieren die Aufrufe dann direkt in die Bedingung:

```
public static void Test_SummeOderProdukt() {
    Console.WriteLine("Möchten Sie multiplizieren (*)
        oder addieren (+)?");
    string operation = Console.ReadLine();
    if (operation == "*") {
        Console.WriteLine("Produkt aus 2 und 3: {0}",
            Produkt(2, 3));
    }
    else {
        Console.WriteLine("Summe aus 2 und 3: {0}",
            Summe(2, 3));
    }
    Console.ReadLine();
}
```

Aufruf per Delegate

Diese beiden Methoden wollen wir nun über eine einzige Methode aufrufen, wobei wir dieser Methode zuvor den Namen der zu verwendenden Methode (also **Summe** oder **Produkt**) übergeben, damit diese weiß, welche Methode sie nutzen soll. Diese gemeinsame Methode heißt dann allerdings nicht mehr Methode, sondern Delegate, und es ist auch keine Methode mehr, sondern ein Objekt. Dieses deklarieren wir wie folgt:

```
public delegate int Berechnung(int a, int b);
```

Die Unterschiede zu einer herkömmlichen Methode sind offensichtlich: Erstens verwenden wir bei der Deklaration das Schlüsselwort **delegate**. Zweitens hat dieses Delegate im Gegensatz zu einer Methode nur eine Deklaration, aber keinen in geschweiften Klammern befindlichen Programmcode. Nun wollen wir über das Delegate **Berechnung** zunächst

die Funktion **Summe** aufrufen. Dazu erstellen wir ein neues Objekt auf Basis des Delegates **Berechnung**. Dabei übergeben wir dem Konstruktor den Namen der Methode, die wir verwenden wollen. Und statt der Methode **Summe()** rufen wir nun das Delegate-Objekt **berechnung** unter Angabe der zu addierenden Zahlen auf:

```
public static void Test_Delegat_Summe() {
    Berechnung berechnung = new Berechnung(Summe);
    Console.WriteLine("Summe aus 2 und 3 per Delegat:
        {0}", berechnung(2, 3));
    Console.ReadLine();
}
```

Auf die gleiche Weise gehen wir vor, wenn wir die Methode **Produkt()** über das Delegate-Objekt **berechnung** aufrufen wollen – mit dem Unterschied, dass wir beim Erstellen des Delegate-Objekts auf Basis des Typs **Berechnung** den Namen der Methode **Produkt** als Konstruktor übergeben:

```
public static void Test_Delegat_Produkt() {
    Berechnung berechnung = new Berechnung(Produkt);
    Console.WriteLine("Produkt aus 2 und 3 per Delegate:
        {0}", berechnung(2, 3));
    Console.ReadLine();
}
```

Berechnung mit Benutzerabfrage und Delegate

Damit können wir nun das erste Beispiel mit der Benutzerabfrage so ändern, dass wir dem Delegate innerhalb der **if...else**-Bedingung die Methode übergeben, die für die Berechnung herangezogen werden soll. Die Berechnung selbst können wir aus der **if...else**-Bedingung herausnehmen. Das Ergebnis sieht nun wie folgt aus:

```
public static void Test_SummeOderProduktMitDelegate() {
    Console.WriteLine("Möchten Sie multiplizieren (*)
        oder addieren (+)?");
    string operation = Console.ReadLine();
    Berechnung berechnung;
    if (operation == "*") {
        berechnung = new Berechnung(Produkt);
    }
    else {
        berechnung = new Berechnung(Summe);
    }
    Console.WriteLine("Ergebnis der Berechnung: {0}",
        berechnung(2, 3));
    Console.ReadLine();
}
```

»Verweis« auf eine Methode

Was ist hier geschehen? Wir haben statt des direkten Aufrufs einer der beiden Methoden **Summe()** beziehungsweise **Produkt()** ein Objekt erschaffen, das einen Zeiger auf die jeweiligen Methoden enthält – eben ein Delegate. Dieses ist prinzipiell ein Objekt, das die Speicheradresse der zu verwendenden Methode enthält. Es erwartet, um die jeweils anvisierte Methode auszuführen, die in der Methode und auch für das Objekt definierten Parameter, also beispielsweise die Summanden beim Addieren oder die Faktoren beim Multiplizieren.

Wichtig ist dabei, dass alle Methoden, die dem Delegate zugewiesen werden, genau die gleichen Parameter aufweisen wie für das Delegate-Objekt festgelegt – in diesem Fall also jeweils zwei **int**-Parameter. Außerdem muss der Datentyp des Rückgabewertes der Methode, soweit vorhanden, mit dem für das Delegate festgelegten Datentyp übereinstimmen.

Delegate ohne Rückgabewert

Angenommen, Sie möchten eine Methode ohne Rückgabewert, wie die Folgende, per Delegate aufrufen:

```
public static void MethodeOhneRueckgabewert() {
    Console.WriteLine("Methode ohne Rückgabewert.");
}
```

Dann verwenden Sie statt der Angabe des Datentyps für den Rückgabewert einfach wie in der Methodendefinition das Schlüsselwort **void**:

```
public delegate void DelegateOhneRueckgabewert();
```

Anwendungskonfigurationsdateien

Während Sie etwa in einer Access-Anwendung leicht die Konfigurationsdaten in einer eigens dafür vorgesehenen Tabelle speichern können oder in einer benutzerdefinierten Text- oder XML-Datei, gibt es unter C# andere Möglichkeiten. Wie für alles finden Sie unter C# auch für das Verwalten von Konfigurationsdaten einen eigenen Namespace – in diesem Fall heißt dieser ConfigurationManager. Dieser Artikel zeigt, wie Sie damit Daten für Desktop-Anwendungen speichern und wieder abrufen können.

Beispielanwendung

Als Beispielanwendung legen wir ein C#-Projekt des Typs Konsolenanwendung an und speichern es unter dem Namen **Konfigurationsdateien**.

Ausprobieren

Wenn Sie die Methoden dieses Artikels selbst in ein neues Projekt eingeben und ausprobieren möchten, fügen Sie diese zur Klasse **Program.cs** hinzu und tragen den Aufruf der Methoden in die Methode **static void Main** ein – etwa wie in folgendem Beispiel:

```
static void Main(string[] args) {
    KonfigurationsdatenLesen
}
```

Alternativ nutzen Sie die im Artikel **Methodenstarter als Vorlage** vorgestellte Variante zum Aufrufen von Beispielmethoden.

Wozu Konfigurationsdateien?

Wozu nutzt man überhaupt Konfigurationsdateien? Darin speichert man Daten, die nicht unbedingt zu den Geschäftsdaten der Anwendung gehören (die in Tabellen landen), sondern die Informationen zur Anwendung wie etwa die Verbindungszeichenfolge enthalten. Diese könnten wir zwar für Anwendungen, die nur auf dem eigenen Rechner laufen und jederzeit neu kompiliert werden können, auch etwa in einer Konstanten speichern. Sobald Sie eine Anwendung jedoch weitergeben möchten, sollten Sie Mechanismen vorsehen, mit denen Konfigurationsdaten an Orten gespeichert werden, die auch der Benutzer zur Laufzeit ändern kann.

Ein gutes Beispiel ist die Verbindungszeichenfolge einer Datenbankanwendung. Ob es sich dabei um eine Verbindung zu einem SQL Server handelt oder auch nur die Angabe des Speicherpfades einer als Backend verwendeten Access-Datenbank, spielt dabei keine Rolle. Tatsache ist, dass es während der Laufzeit geschehen kann, dass der Benutzer diese Daten ändern können sollte, um beispielsweise den Servernamen einer SQL Server-Verbindung zu ändern oder auch einen neuen Speicherpfad einer Access-Datenbank einzugeben.

Eine solche Konfigurationsdatei mit einer Verbindungszeichenfolge für eine Access-Datenbank sieht beispielsweise wie folgt aus:

```
<?xml version="1.0" encoding="utf-8"?>
<configuration>
  <connectionStrings>
    <add name="Accessdatenbank"
        connectionString="c:\Suedsturm.mdb" />
  </connectionStrings>
</configuration>
```

Speicherort der Konfigurationsdateien

Für eine C#-Anwendung legt Visual Studio drei Konfigurationsdateien an, die Sie im Projektverzeichnis finden. Direkt im Hauptverzeichnis landet die Datei **App.config**. Im Verzeichnis **bin\Debug** finden Sie beim Debuggen die beiden Dateien **<Anwendungsname>.exe.config** und **<Anwendungsname>.vshost.exe.config**. Später, wenn Sie die Anwendung erstellen, ist zunächst keine **.config**-Datei vorhanden. Diese wird erstellt, sobald die Anwendung Konfigurationsinforma-

tionen speichert – und zwar ebenfalls unter dem Namen **<Anwendungsname>.exe.config**.

Wenn Sie eine Anwendungskonfigurationsdatei zu einem Projekt hinzufügen möchten, das nach der Installation bereitsteht, legen Sie die gewünschte Datei im XML-Format an und speichern diese im Anwendungsverzeichnis.

Konfigurationsdatei manuell hinzufügen

Wenn Sie eine Konfigurationsdatei hinzufügen möchten, statt diese per Code zu erzeugen, erledigen Sie dies am einfachsten über den Menüeintrag **Projekt|Neues Element hinzufügen**. Dies öffnet den Dialog **Neues Element hinzufügen**, wo Sie den Eintrag **Anwendungskonfigurationsdatei** auswählen (siehe Bild 1). Geben Sie den gewünschten Namen für die Anwendungskonfigurationsdatei ein und klicken Sie auf die Schaltfläche **Hinzufügen**.

Die neue Anwendungskonfigurationsdatei sieht nun wie in Bild 2 aus und wird auch im Projektmappen-Explorer angezeigt. Prinzipiell benötigen Sie aber keine

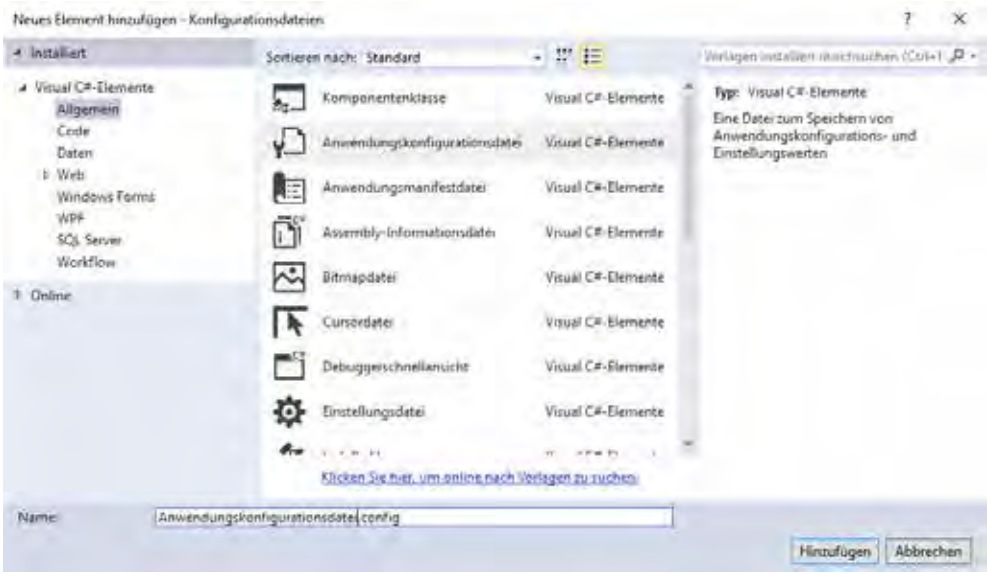


Bild 1: Anlegen einer Anwendungskonfigurationsdatei

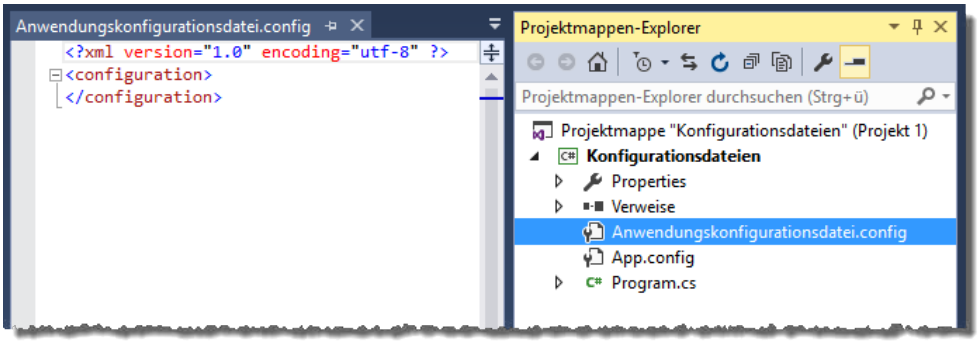


Bild 2: Die neue Anwendungskonfigurationsdatei

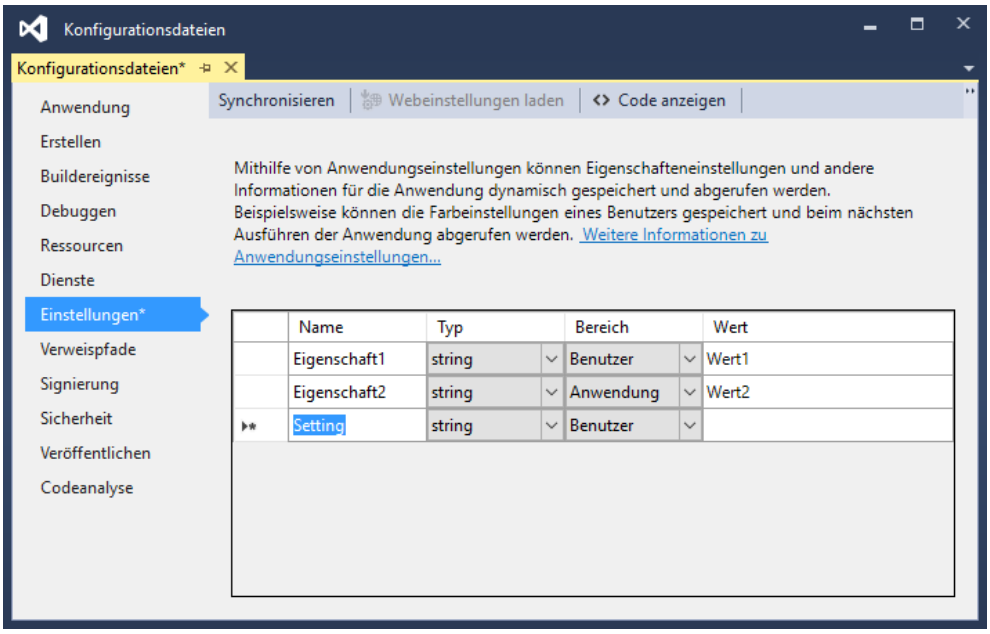


Bild 3: Einstellungen zu einer Anwendung hinzufügen

```
<?xml version="1.0" encoding="utf-8" ?>
<configuration>
    <configSections>
        <sectionGroup name="userSettings" ... >
            <section name="Konfigurationsdateien.Properties.Settings" ... />
        </sectionGroup>
        <sectionGroup name="applicationSettings" ... >
            <section name="Konfigurationsdateien.Properties.Settings" ... />
        </sectionGroup>
    </configSections>
    <startup>
        <supportedRuntime version="v4.0" sku=".NETFramework,Version=v4.5.2" />
    </startup>
    <userSettings>
        <Konfigurationsdateien.Properties.Settings>
            <setting name="Eigenschaft1" serializeAs="String">
                <value>Wert1</value>
            </setting>
        </Konfigurationsdateien.Properties.Settings>
    </userSettings>
    <applicationSettings>
        <Konfigurationsdateien.Properties.Settings>
            <setting name="Eigenschaft2" serializeAs="String">
                <value>Wert2</value>
            </setting>
        </Konfigurationsdateien.Properties.Settings>
    </applicationSettings>
</configuration>
```

Listing 1: Konfigurationsdatei **app.config** mit einer Benutzer- und einer Anwendungseinstellung

neue Konfigurationsdatei, denn das Projekt bringt mit **app.config** bereits eine solche mit. Sollten Sie diese Datei jedoch einmal versehentlich löschen, können Sie diese mit der oben beschriebenen Vorgehensweise wieder herstellen.

Konfigurationsdaten anlegen

Um Konfigurationsdateien mit Eigenschaften auszustatten, verwenden Sie den Einstellungen-Editor von Visual Studio. Um diesen zu öffnen, klicken Sie doppelt auf den Eintrag **Properties** im Projektmappen-Explorer. Wechseln Sie dann zum Bereich **Einstellungen** und fügen Sie die benötigten Eigenschaften samt Datentyp, Bereich (**Benutzer** oder **Anwendung**) sowie den Standardwert fest (siehe Bild 3). Warum Standardwert? Weil die Konfigurationsdatei erst beim Erstellen der Anwendung erzeugt und mit Werten gefüllt wird. Um dies zu prüfen, erstellen Sie die Anwendung zum Debuggen (**F5**) und schauen sich anschließend den Inhalt der Datei **app.config** im Hauptverzeichnis des Projekts an. Diese sieht nun in gekürzter Form wie in Listing 1 aus. Im Bereich **userSettings** finden Sie die Eigenschaft mit der Bezeichnung **Eigenschaft1** und dem Wert **Wert1**, unter **applicationSettings** die Eigenschaft namens **Eigenschaft2** mit dem Wert **Wert2**.

Einlesen der Konfigurationsdateien

Wenn Sie die so angelegten Einstellungen von einer C#-Anwendung aus einlesen möchten, erledigen Sie das etwa mit einer Methode wie der folgenden:

```
public static void KonfigurationsdatenLesen() {
    Settings settings = new Settings();
```

```
    Console.WriteLine("Wert von Eigenschaft1: {0}",
        settings.Eigenschaft1);
    Console.WriteLine("Wert von Eigenschaft2: {0}",
        settings.Eigenschaft2);
    Console.ReadLine();
}
```

Bei der Eingabe dieser Methode wird Visual Studio meckern, weil es das Objekt **Settings** nicht kennt. **Settings** ist eine Klasse, die automatisch angelegt wird, sobald Sie im Bereich **Properties** der Anwendungseinstellungen eine Eigenschaft angelegt haben. Diese Klasse sehen Sie auch im Projektmappen-Explorer (siehe Bild 4). Damit Sie im Code komfortabel auf die Elemente der **Settings**-Klasse zugreifen können, fügen Sie dem Klassenmodul mit der **using**-Direktive einen Verweis auf das Element **<Anwendungsname>.Properties** hinzu:

```
using Konfigurationsdateien.Properties;
```

Danach ist der Zugriff auf die Klasse **Settings** per Code problemlos möglich, und Sie können die Werte der per Benutzeroberfläche hinzugefügten Einstellungen über **settings.Eigenschaft1** und **settings.Eigenschaft2** ausgeben. Die Klasse **Settings** wird automatisch bei Änderungen an den Eigenschaften neu erstellt und arbeitet wie eine benutzerdefinierte Klasse – sie stellt die Eigenschaften per **Set**- und **Get**-Methode zum Lesen und Schreiben zur Verfügung. Ein Blick in den Code dieser Klasse zeigt jedoch direkt, dass die Einstellungen des Bereichs **Benutzer** gelesen und geändert, die Einstellungen des Bereichs **Anwendung** jedoch nur gelesen werden können.

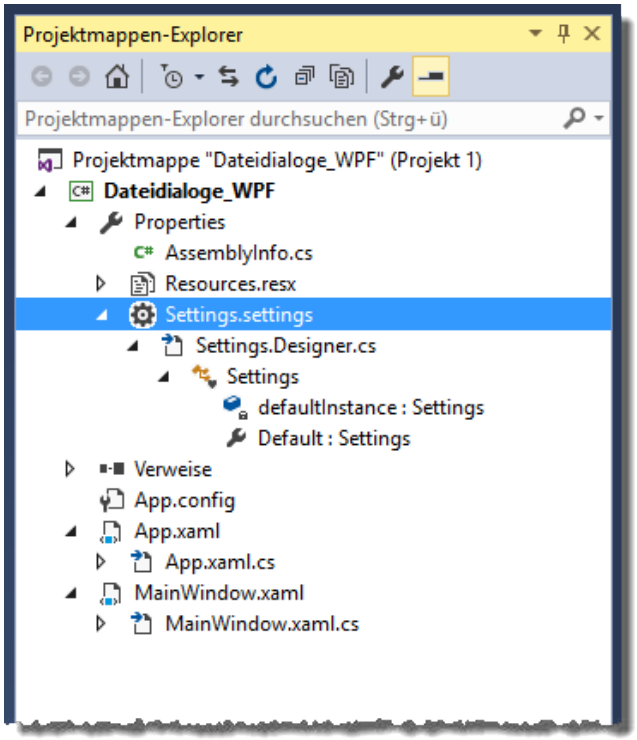


Bild 4: Die Klasse **Settings** stellt die benutzerdefinierten Einstellungen bereit.

Benutzereinstellungen ändern

Wenn Sie also per Code Einstellungen ändern möchten, gelingt dies ohne Weiteres nur bei den Benutzereinstellungen – und zwar beispielsweise wie folgt:

```
public static void KonfigurationsdatenAendern() {
    Settings settings = new Settings();
    settings.Eigenschaft1 = "Test1_Neu";
    settings.Save();
    Console.ReadLine(); }
```

Dies ändert den Wert von **Eigenschaft1** auf **Test1_Neu**. Wenn Sie diese Methode aufrufen und anschließend nochmals die Methode **KonfigurationsdatenLesen**, wird der neue Wert ausgegeben.

Speicherort nach der Weitergabe

Wo sich die Konfigurationsdatei beim Debuggen befindet, haben wir bereits weiter oben erläutert. Aber wo finden Sie die entsprechenden Dateien, wenn Sie die **.exe**-Datei weitergeben und auf einem anderen Rechner ausführen? Genau genommen handelt es sich in diesem Fall um zwei Konfigurationsdateien. Die erste enthält die Benutzereinstellungen, die zweite die Anwendungseinstellungen. Die Benutzereinstellungen finden Sie etwa unter Windows 7 in einem Verzeichnis, das sich wie folgt zusammensetzt:

```
C:\Users\<Benutzername>\AppData\Local\<Hersteller>\<Anwendungsname>.exe_<Evidence Type>_<Evidence Hash>\<Version>\user.config
```

Wenn Sie für **<Hersteller>** keine Angabe in der Datei **AssemblyInfo.cs** gemacht haben, verwendet die Anwendung die Bezeichnung des Namespace (hier **nsKonfigurationsdateien**). **<Evidence Type>** und **<Evidence Hash>** werden vom System generiert. Diese Datei enthält nur den Bereich **userSettings** der weiter oben vorgestellten Konfigurationsdatei:

```
<configuration>
    <userSettings>
        <Konfigurationsdateien.Properties.Settings>
            <setting name="Eigenschaft1" serializeAs="String">
                <value>Test1_Neu</value>
            </setting>
        </Konfigurationsdateien.Properties.Settings>
    </userSettings>
</configuration>
```

Wie nun können wir etwa per Code ermitteln, wo sich diese Datei befindet? Dazu fügen Sie zunächst einen Verweis auf **System.Configuration** zum Projekt hinzu (siehe Bild 5) und erstellen dann Sie ein Objekt des Typs **Configuration**:

```
Configuration config =
    ConfigurationManager.OpenExeConfiguration(
    ConfigurationUserLevel.PerUserRoamingAndLocal);
```

Dieses liefert mit der Eigenschaft **FilePath** den Pfad zur Konfigurationsdatei:

```
Console.WriteLine("Speicherort der
    Benutzerkonfigurationsdatei: {0}",
    config.FilePath);
```

Diese beiden Zeilen können Sie beispielsweise in der Methode **KonfigurationsdatenLesen** unterbringen.

Ändern per Konfigurationsdatei

Sie oder der Benutzer können die Einstellungen der Datei **user.config** natürlich auch direkt in der Datei ändern. Diese stehen anschließend direkt in der Anwendung zur Verfügung.

Anwendungseinstellungen ändern

Wie Sie weiter oben erfahren haben, können Sie Einstellungen im Bereich **applicationSettings** nicht wie die im Bereich **userSettings** per Code ändern. Wie aber können wir dennoch anwendungsweite Einstellungen pflegen, die alle Benutzer betreffen? Dazu gibt es einen weiteren Bereich namens **appSettings**. Die Inhalte dieses Bereichs können Sie nicht so einfach über die Benutzeroberfläche von Visual Studio vorbereiten wie bei den beiden Bereichen **userSettings** und **applicationSettings**, die Sie weiter oben kennen gelernt haben. In der **.config**-Datei sieht der Bereich wie folgt aus, hier mit zwei Beispieleigenschaften:

```
<configuration>
    <configSections>...</configSections>
    ...
    <appSettings>
        <add key="eigenschaft1" value="wert1" />
        <add key="eigenschaft2" value="wert2" />
    </appSettings>
    ...
</configuration>
```

Die Einträge dieses Bereichs lesen Sie beispielsweise mit der Methode aus Listing 2. Die Methode verwendet die Eigenschaft **Count** der Auflistung **AppSettings** des **ConfigurationManager**-Objekts, um die Anzahl der Einstellungen zu ermitteln und in der Variablen **count** zu speichern. Ist **count**

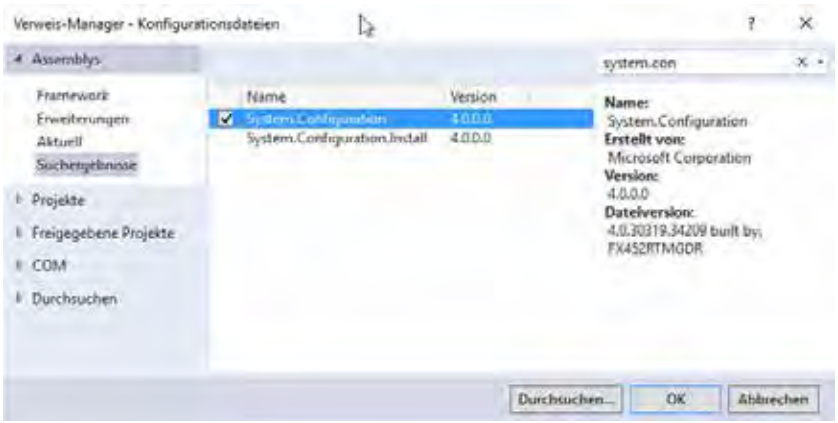


Bild 5: Die Klasse **Settings** stellt die benutzerdefinierten Einstellungen bereit.

Datenzugriff mit ADO.NET, Teil 2

Im ersten Teil unserer Artikelreihe zum Thema ADO.NET haben wir uns angesehen, wie Sie mit dem DataReader-Objekt auf die Daten einer Tabelle zugreifen und wie Sie mit dem Command-Objekt Aktionsabfragen durchführen können. Im vorliegenden Teil sehen wir uns an, wie Sie mit dem DataAdapter-Objekt auf die Daten einer Datenbank zugreifen und das DataSet- und das DataTable-Objekt einsetzen, um verbindungslos mit den Daten einer Datenbank zu arbeiten.

Die im ersten Teil dieser Artikelreihe vorgestellten Techniken für den Zugriff auf die Daten einer Datenbank erforderten eine geöffnete Verbindung. So etwas ist in einer Desktopdatenbank etwa auf Basis von Microsoft Access kein Problem. Etwas kritischer wird es schon, wenn Sie Access in einer Mehrbenutzerumgebung einsetzen – und vor allem dann, wenn die Anwendung über das Internet über die anzuzeigenden oder zu bearbeitenden Daten zugreift. In Zeiten der Mehrbenutzer-, Internet- und mobilen Anwendungen benötigen wir daher verbindungslose Zugriffstechnologien.

Die unterstützt ADO.NET durch einige weitere Objekte, welche nach Wunsch mit den Daten einer oder mehrerer Tabellen oder Abfragen der Datenbank gefüllt werden. Der Benutzer zeigt die Daten an, löscht, bearbeitet oder erweitert sie und stößt dann den Speichervorgang für die Daten an. Daraufhin wird erneut eine Verbindung geöffnet und die geänderten Daten werden zurück in die Datenbank geschrieben. Wie dies gelingt, schauen wir uns weiter unten an. Vorher jedoch noch einige Ergänzungen zum ersten Teil dieser Artikelreihe.

Verbindungszeichenfolge einfach festlegen

Wer sich mit dem Zusammenstellen der Zeichenkette für die Verbindungszeichenfolge schwer tut und/oder Eingabefehler bei der Benennung der Parameter vermeiden will, kann auch ein Objekt namens **ConnectionStringBuilder** verwenden. Dieses gibt es ebenso wie die Objekte **Connection**, **Command** et cetera jeweils mit entsprechendem Präfix für die verschiedenen ADO.NET-Klassen wie **System.Data.OLEDB**, **System.Data.ODBC**, **System.Data.SQL** und so weiter. Da wir aktuell noch eine Access-Datenbank als Datenquelle für unsere Beispiele nutzen, heißt die entsprechende Klasse also **OLEDBConnectionStringBuilder**. Die folgende Prozedur erstellt ein neues Objekt auf Basis dieser Klasse und speichert es in der Variablen **ConnectionStringBuilder**. Die Objektvariable stellt alle für die aktuelle Verbindung möglichen Parameter als Eigenschaften zur Verfügung, sodass Sie die gewünschten Elemente einfach per IntelliSense auswählen wollen.

Im vorliegenden Fall weisen wir so den Parameter **DataSource** und den Parameter **Provider** zum Objekt **ConnectionStringBuilder** hinzu und stellen so die Verbindungszeichenfolge zusammen, die wir dann über die Eigenschaft **ConnectionString** abfragen und beim Erstellen des **OLEDBConnection**-Objekts verwenden können:

```
public static void ConnectionstringBuilder() {
    OleDbConnectionStringBuilder ConnectionStringBuilder =
        new OleDbConnectionStringBuilder();
    ConnectionStringBuilder.DataSource = "Suedsturm.mdb";
    ConnectionStringBuilder.Provider =
        "Microsoft.Jet.OLEDB.4.0";
    OleDbConnection cnn = new OleDbConnection(
        ConnectionStringBuilder.ConnectionString);
    try {
        cnn.Open();
        Console.WriteLine("Provider: {0}", cnn.Provider);
        Console.WriteLine("DataSource: {0}",
            cnn.DataSource);
        Console.ReadLine();
        cnn.Close();
    }
```

```
    catch(Exception e) {
        Console.WriteLine("Fehler: {0}", e.Message);
        Console.ReadLine();
    }
}
```

Im Folgenden haben wir die Ausgabe einiger Eigenschaften des **Connection**-Objekts in eine rudimentäre Fehlerbehandlung gesteckt. Diese greift, wenn innerhalb des **try**-Blocks ein Fehler auftritt. Dann wird der **catch**-Block ausgelöst, der eine Fehlermeldung in der Konsole ausgibt. Mehr zum Thema Fehlerbehandlung lesen Sie im Artikel **Fehlerbehandlung mit C#**.

Verbindungszeichenfolge speichern

Wenn Sie unter Access eine Backend-Datenbank verwenden, wird der Pfad zu den verknüpften Tabellen in der Systemtabelle **MSysObjects** gespeichert. Sollten die verknüpften Tabellen nicht an der vorgesehen Stelle verfügbar sein, löst dies einen Fehler aus. Dies fängt man unter Access ab, indem man beim Öffnen der Datenbank prüft, ob die verknüpfte Datenbank sich am vorgesehenen Ort befindet. Falls nicht, blendet man einen Dialog ein, der dem Benutzer die Auswahl des neuen Speicherorts der Backend-Datenbank ermöglicht und verknüpft die Tabellen dann erneut.

Dieses Verhalten wollen wir unter C# für Desktop-Anwendungen ähnlich abbilden – zumindest so, dass es für den Benutzer so aussieht. Intern läuft es natürlich etwas anders: Es gibt ja keine Verknüpfung zu einer Datenbank, sondern wir verwenden eine Zeichenkette, welche den Speicherort der zu nutzenden Datenbank angibt. Zumindest in den bisherigen Beispielen (und der Einfachheit halber auch in den meisten weiteren) legen wir die Datenbank direkt im Verzeichnis der **.exe**-Datei ab. Spätestens, wenn Sie einmal eine Anwendung für den Mehrbenutzerbetrieb auslegen und dabei eine Datei wie eine Access-Datenbank, aber möglicherweise auch eine Excel- oder XML-Datei als Datenquelle nutzen, sollten Sie eine Möglichkeit vorsehen, das Vorhandensein der Quelldatei zu prüfen und gegebenenfalls den Speicherort der Datei neu zu ermitteln.

Hierzu besteht die erste Aufgabe darin, den Pfad zur Quelldatei an einem Ort zu speichern, der außerhalb der kompilierten Anwendung liegt. Anderenfalls müssten Sie die Anwendung ja jedes Mal neu kompilieren, wenn sich der Pfad ändert. Das können Sie bei selbst genutzten Anwendungen tun, aber sicher nicht bei solchen Anwendungen, die Sie an Kunden oder andere Benutzer weitergeben.

Unter Access hätten wir für einen solchen Zweck einfach eine lokale Optionentabelle verwendet oder alternativ eine Textdatei. Eine C#-Desktop-Anwendung jedoch enthält ja keine eigenen Tabellen zum Speichern von Daten, also müssen wir schon auf eine Art Konfigurationsdatei zurückgreifen. Es gibt jedoch gute Nachrichten: C# sieht für Desktop-Anwendungen eine Möglichkeit vor, Konfigurationsdateien zu pflegen und die enthaltenen Daten einfach per Code einzulesen und auch zu ändern.

Wie Sie die Verbindungszeichenfolge in einer Konfigurationsdatei speichern, erfahren Sie im Artikel **Anwendungskonfigurationsdateien**.

Der DataAdapter

Genau wie die übrigen ADO.NET-Objekte gibt es auch den **DataAdapter** wieder in verschiedenen Klassen. Dementsprechend heißt der **DataAdapter** beispielsweise **SqlDataAdapter**, **OdbcDataAdapter** oder **OleDbDataAdapter** – je nachdem, mit welcher Klasse Sie auf welchen Datenbanktyp zugreifen möchten.

Wir beschäftigen uns der Einfachheit halber immer noch mit unserer Access-Beispieldatenbank **Suedsturm.mdb**, also nutzen wir den **OleDbDataAdapter**. Im Folgenden reden wir jedoch neutral von **DataAdapter**, nur im Beispielcode finden Sie entsprechend die Objektbezeichnung **OleDbDataAdapter**.

Was bietet uns der **DataAdapter**? Mit dem **Command**- und dem **DataReader**-Objekt konnten wir ja immerhin schon einmal Auswahlabfragen ausführen und die Ergebnisse vorwärts durchlaufen oder Aktionsabfragen anstoßen. Der

DataReader erlaubt es aber beispielsweise nicht, einen der enthaltenen Datensätze zu ändern und die geänderte Version zurück in die zugrunde liegenden Tabellen zu schreiben. Genauso wenig, wie Sie damit durch die Datensätze navigieren können – es geht nur vorwärts.

Außerdem sind **Command** und **DataReader** Objekte, die eine Verbindung zur Datenbank benötigen. Gerade dies ist natürlich für Web-Anwendungen oder Anwendungen auf mobilen Endgeräten nicht sinnvoll, und auch für Desktop-Anwendungen in Mehrbenutzerumgebungen können nicht endlos viele Verbindungen gleichzeitig offen gehalten werden. Daher sieht ADO.NET einige Objekte vor, mit denen Sie die Daten aus der Datenbank einlesen, bearbeiten und wieder zurückschreiben können. Dabei ist nur für das Einlesen und das Zurückschreiben eine Verbindung nötig, zwischendurch wird die Verbindung getrennt.

Und hier kommt der **DataAdapter** ins Spiel (je nach Anwendungsfall als **SqlDataAdapter**, **OdbcDataAdapter**, **OleDbDataAdapter** et cetera): Er stellt die Verbindung zwischen Client und Server her, um die benötigten Daten aus den Tabellen der Datenbank in lokale Objekte zu übertragen und diese nach Änderungen wieder zurückzuschicken. Dabei stellt der **DataAdapter** nur Verbindungen zum Server her, wenn tatsächlich Daten bewegt werden müssen.

Connection- und Command-Objekte

Die beiden Objekte **Connection** und **Command**, die Sie bereits kennen gelernt haben, dürfen Sie jedoch nicht aus Ihrem Gedächtnis streichen: Das **Connection**-Objekte benötigen wir nämlich nach wie vor, auch um die Verbindungen des **DataAdapters** herzustellen.

Und das **Command**-Objekt ist immer noch gefragt, wenn es darum geht, Änderungen an den Daten einer Tabelle vorzunehmen, für die wir die vorhandenen Daten nicht erst einlesen und analysieren müssen. Eine Aktionsabfrage etwa zum Anfügen eines neuen Datensatzes oder zum Löschen oder Bearbeiten vorhandener Datensätze ist per **Command**-Objekt immer noch schneller, als wenn Sie die entsprechenden Tabellen über den **DataAdapter** in weitere Objekte laden, die Daten dort ändern und die Änderungen dann zurückschreiben.

DataTable und DataRow

Um mit den über den **DataAdapter** gewonnenen Daten zu arbeiten und etwa in einer Schleife alle Datensätze zu durchlaufen und anzuzeigen, benötigen Sie noch entsprechende Objekte, um diese zu speichern. Das **DataTable**-Objekt nimmt dabei das Ergebnis einer Tabelle oder Abfrage auf. Seine **Rows**-Eigenschaft erlaubt den Zugriff auf die einzelnen Datensätze. Diese können Sie wiederum mit dem **DataRow**-Objekt referenzieren, um gezielt auf die Inhalte zuzugreifen.

Im Gegensatz zu den Objekten **Connection**, **Command** und **DataReader**, die ja je nach Anforderung etwa aus einem der Namespaces **System.Data.Sql**, **System.Data.Odbc** oder **System.Data.OleDb** stammen, kommen die beiden Objekte **DataTable** und **DataRow** aus dem Namespace **System.Data**. Diesen fügen Sie also noch über folgende Anweisung zur Klasse hinzu:

```
using System.Data;
```

DataTable füllen und durchlaufen

Damit kommen wir nun endlich zum ersten handfesten Beispiel dieses Artikels. Dabei wollen wir ein **DataTable**-Objekt über einen **DataAdapter** mit den Daten einer Tabelle füllen und die Datensätze durchlaufen und in der Konsole ausgeben (siehe Listing 1).

Die Methode speichert zunächst die Verbindungszeichenfolge und die SQL-Abfrage in den beiden Variablen **strConnection** und **strSQL**.

Dann erzeugt sie ein neues Objekt des Typs **OleDbConnection** und übergibt dabei gleich die zu verwendende Verbindungszeichenfolge, mit der in diesem Beispiel auf die Access-Datenbank **Suedsturm.mdb** im Verzeichnis der C#-Anwendung selbst zugegriffen werden soll.

```
public static void DataTableFuellen() {
    string strConnection = "Provider=Microsoft.Jet.OLEDB.4.0;Data Source=Suedsturm.mdb";
    string strSQL = "SELECT * FROM tblKategorien";
    OleDbConnection cnn = new OleDbConnection(strConnection);
    OleDbDataAdapter da = new OleDbDataAdapter(strSQL, cnn);
    DataTable dt = new DataTable();
    da.Fill(dt);
    for (int i = 0; i < dt.Rows.Count; i++) {
        DataRow row = dt.Rows[i];
        Console.WriteLine("{0} {1}", row[0], row[1]);
    }
    Console.ReadLine();
}
```

Listing 1: Füllen und Durchlaufen einer Tabelle per **DataTable**

Den **OleDbDataAdapter** speichert die Methode nach dem Erstellen und der Übergabe der SQL-Anweisung mit **strSQL** und dem **Connection**-Objekt in der Variablen **da**.

Hier fällt auf, dass die Methode die Verbindung nicht explizit öffnet, indem Sie die Open-Methode aufruft. Dies ist nicht nötig, da die Fill-Methode des DataAdapter-Objekts diese Aufgabe für uns übernimmt. Sollten Sie für andere Aktionen in diesem Zusammenhang eine offene Verbindung benötigen, können Sie diese mit Open öffnen, müssen dann aber auch mit Close für das Schließen der Verbindung sorgen.

Das **DataTable**-Objekt erzeugt die Methode zunächst einfach nur und speichert es in der Variablen **dt**. Das Füllen dieses Objekts erfolgt nicht über eine Methode des Objekts selbst, sondern das Objekt wird als Parameter der **Fill**-Methode des **DataAdapter**-Objekts übergeben:

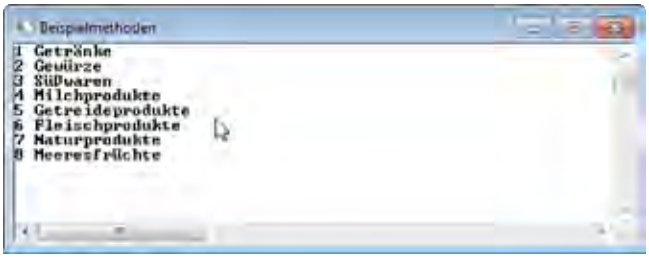


Bild 1: Ausgabe der Daten der Tabelle **tblKategorien**

```
da.Fill(dt);
```

Nun enthält das **DataTable**-Objekt bereits alle Datensätze mit allen Feldern der Tabelle **tblKategorien**. Diese durchläuft die Methode in einer **for**-Schleife mit der Laufvariablen **i**, welche die Werte von **0** bis zur Anzahl der Datensätze minus eins durchläuft. Die Anzahl der Datensätze ermitteln wir hier mit dem Ausdruck **dt.Rows.Count**. **Rows** ist die Auflistung aller Datensätze.

Über diese Auflistung erhalten wir mit dem entsprechenden Indexwert ein **DataRow**-Objekt, das wir mit jedem Schleifendurchlauf neu mit der Variablen **row** referenzieren:

```
DataRow row = dt.Rows[i];
```

Über den Index der Felder gibt die Methode nun die Inhalte der ersten beiden Felder in der Konsole aus:

```
Console.WriteLine("{0} {1}", row[0], row[1]);
```

In dieser Anweisung könnten wir auch konkret auf die Feldnamen zugreifen:

```
Console.WriteLine("{0} {1}", row["KategorieID"],
row["Kategoriename"]);
```

Anschließend gibt die Methode die Liste der Kategorien aus der Tabelle **tblKategorien** in der Konsole aus (siehe Bild 1).

Wir haben nun das **OleDbConnection**-Objekt und die SQL-Anweisung per Konstruktor an das **OleDbDataAdapter**-Objekt übergeben. Sie können auch zunächst ein **OleDbCommand**-Objekt erstellen und dieses dann als Konstruktor beim Erstellen des **OleDbDataAdapter**-Objekts übergeben:

```
string strConnection = "Provider=Microsoft.Jet.
OLEDB.4.0;Data Source=Suedsturm.mdb";
string strSQL = "SELECT * FROM tblKategorien";
OleDbConnection cnn = new OleDbConnection(strConnection);
OleDbCommand cmd = new OleDbCommand(strSQL, cnn);
OleDbDataAdapter da = new OleDbDataAdapter(cmd);
```

Oder Sie verwenden beim Erstellen des **DataAdapter**-Objekts die konstruktorlose Variante und weisen das **OleDbCommand**-Objekt erst im Anschluss über die Eigenschaft **SelectCommand** hinzu:

```
...
OleDbCommand cmd = new OleDbCommand(strSQL, cnn);
OleDbDataAdapter da = new OleDbDataAdapter();
da.SelectCommand = cmd;
```

Warum ist hier von **SelectCommand** die Rede, und nicht einfach von **Command**? Weil der **DataAdapter** ja zunächst einmal Daten liefern soll. Daher soll das **Command**-Objekt eine **Select**-Anweisung enthalten und nicht etwa eine Aktionsabfrage.

Anzahl der Datensätze per Fill

Wenn Sie ein **DataTable**-Objekt mit der **Fill**-Methode des **DataAdapter**-Objekts füllen, liefert die **Fill**-Methode einen Rückgabewert, den Sie hier direkt nutzen können: Es handelt sich nämlich um die Anzahl der zurückgelieferten Datensätze. Im folgenden Ausschnitt der Methode **DataTableFuellenIV** schreiben wir das Ergebnis der **Fill**-Methode in die Variable **anzahl**, die wir dann als Vergleichswert des zweiten Parameters der **for**-Schleife nutzen:

```
...
DataTable dt = new DataTable();
int anzahl = da.Fill(dt);
for (int i = 0; i < anzahl; i++)
{
    DataRow row = dt.Rows[i];
    Console.WriteLine("{0} {1}", row[0], row[1]);
}
...
```

DataTable per foreach durchlaufen

Das Durchlaufen der Datensätze des **DataTable**-Objekts ist eine Variante. Etwas eleganter sieht es mit der **foreach**-Schleife aus. Dazu verwenden wir ein **DataRow**-Objekt als Laufvariable und ermitteln dieses jeweils aus der **Rows**-Auflistung des **DataTable**-Objekts:

```
da.Fill(dt);
foreach (DataRow datensatz in dt.Rows) {
    Console.WriteLine("{0} {1}", datensatz[0],
        datensatz[1]);
}
```

Das DataSet-Objekt

Ein **DataTable**-Objekt nimmt das Ergebnis einer SQL-Abfrage auf. Was aber, wenn man mehrere verknüpfte Tabellen referenzieren und deren Inhalte anzeigen, bearbeiten, ergänzen oder löschen möchte? Wenn Sie etwa einen Artikel zur Tabelle **tblArtikel** hinzufügen möchten und gleichzeitig eine neue Kategorie für diesen Artikel in der Tabelle **tblKategorien** anlegen möchten, können sie dies natürlich über zwei einzelne **DataTable**-Objekte realisieren. Wenn Sie jedoch deren Beziehung automatisch berücksichtigen möchten, nutzen Sie das **DataSet**-Objekt, das ein oder mehrere Tabellen berücksichtigt.

Typisierte und untypisierte DataSets

DataSets kommen in zwei Ausführungen: Im Folgenden besprechen wir zunächst die sogenannten »untypisierten« DataSets. Diese liefern nur die Standardmethoden, -eigenschaften und -ereignisse und sind mit einem Recordset

unter Access vergleichbar. Wenn Sie etwa die Felder referenzieren möchten, um darauf zuzugreifen, geben Sie entweder den Index oder den Namen des jeweiligen Feldes als Parameter in eckigen Klammern an. Unter Access hätten Sie etwa für den Zugriff auf das Feld **Kategoriename** der Tabelle **tblKategorien** in einem Recordset namens **rstKategorien** den Ausdruck **rstKategorien!Kategoriename** oder **rstKategorien("Kategoriename")** verwendet. Sie hätten aber nicht etwa per IntelliSense das Feld **Kategoriename** als Eigenschaft des Objekts **rstKategorien** eingeben können.

So ähnlich ist es bei untypisierten **DataSet**-Objekten: Hier greifen Sie etwa über den Ausdruck **KategorieRow[0]["Kategoriename"]** auf das Feld **Kategoriename** zu. Bei einem typisierten **DataSet**-Objekte legt Visual Studio für Sie eine Klassenstruktur an, die es ermöglicht, per IntelliSense auf die Felder der Tabelle zuzugreifen. Hier sähe der Zugriff dann etwa so aus: **KategorieRow[0].Kategoriename**. Damit erhalten Sie Typsicherheit: Sie können so zum Beispiel nicht etwa durch einen Schreibfehler einen falschen Feldnamen angeben, da die Feldnamen über eine Klasse alle fest definiert sind.

DataSet mit einer Tabelle füllen

Wir schauen uns zunächst an, wie dies mit einer einzelnen Tabelle funktioniert (siehe Listing 2). Dazu verwenden wir wieder die gleiche Verbindungszeichenfolge und SQL-Anweisung wie zuvor, und erstellen ein **Connection**-Objekt und ein **DataAdapter**-Objekt. Dann jedoch legen wir ein neues **DataSet**-Objekt namens **ds** an, dem wir keinen Konstruktor übergeben. Wir füllen es genauso wie das **DataTable**-Objekt, indem wir es der Methode **Fill** des **DataAdapter**-Objekts übergeben:

```
DataSet ds = new DataSet();
int anzahl = da.Fill(ds);
```

Die **Fill**-Methode liefert die korrekte Anzahl der Datensätze zurück, die wir dann in der **for**-Schleife zum Durchlaufen der Datensätze verwenden. Hier gibt es nun in Ermangelung des **DataTable**-Objekts eine kleine Änderung: Wir greifen nämlich auf das erste Element der **Tables**-Auflistung des **DataSet**-Objekts zu und erhalten wiederum genau ein **DataTable**-Objekt, auf dessen Feldinhalte wir über den Index zugreifen können:

```
DataRow row = ds.Tables[0].Rows[i];
Console.WriteLine("{0} {1}", row[0], row[1]);
```

Wir haben hier also den Fall abgebildet, dass ein **DataSet**-Objekt genau ein **DataTable**-Objekt enthält.

Später kommen wir darauf zurück, wie Sie mehrere verknüpfte Tabellen in ein **DataSet** einbetten.

```
public static void DataSetFuellenI() {
    string strConnection = "Provider=Microsoft.Jet.OLEDB.4.0;Data Source=Suedsturm.mdb";
    string strSQL = "SELECT * FROM tblKategorien";
    OleDbConnection cnn = new OleDbConnection(strConnection);
    OleDbDataAdapter da = new OleDbDataAdapter(strSQL, cnn);
    DataSet ds = new DataSet(); //DataSet statt DataTable
    int anzahl = da.Fill(ds);
    for (int i = 0; i < anzahl; i++) {
        DataRow row = ds.Tables[0].Rows[i]; //Deshalb hier Zugriff auf ds.Tables[0] statt auf dt wie zuvor
        Console.WriteLine("{0} {1}", row[0], row[1]);
    }
    Console.ReadLine();
}
```

Listing 2: Füllen und Durchlaufen einer Tabelle per **DataSet**

Fenster mit einfachen Tabellendaten

Unter Access haben Sie mit wenigen Mausklicks die Daten einer Tabelle in einem Formular angezeigt – Datenherkunft festlegen, Felder aus der Feldliste in das Formular zeihen, fertig war das Formular. Unter WPF ist es erstens etwas aufwendiger, und zweitens gibt es verschiedene Möglichkeiten, dies zu realisieren. In diesem Artikel schauen wir uns die einfachste Methode an, die Daten einer einfachen Tabelle in einem Fenster anzuzeigen, in den Datensätzen zu navigieren und Änderungen zu speichern.

Wie unter Access gibt es auch unter C# einfache und schnelle Wege, ein Ziel zu erreichen – nämlich mit Unterstützung der Entwicklungsumgebung, meist in Form von Assistenten. Wer später Änderungen am Entwurf seiner Benutzeroberfläche vornehmen möchte, sollte jedoch wissen, was der Assistent dort im Einzelnen gezaubert hat.

Deshalb schauen wir uns in diesem Artikel zunächst an, wie Sie die Daten einer Tabelle so schnell wie möglich in einem Fenster anzeigen. Später erfahren Sie dann, was dort alles geschehen ist und wie Sie ein solches Fenster von Hand erstellen und auch anpassen können.

Voraussetzung

Voraussetzung für die Beispiele dieses Artikels ist das Vorhandensein der Beispieldatenbank **Suedsturm.mdb**, einer Access-Datenbank. Diese sollte im Projektverzeichnis liegen und auch dem Projekt im Projektmappen-Explorer zugewiesen sein, damit dieses beim Debuggen automatisch in den **Debug**-Ordner kopiert wird und dort für den Zugriff beim Testen zur Verfügung steht. Ist dies nicht der Fall, ziehen Sie die Datenbankdatei **Suedsturm.mdb** einfach aus dem Windows Explorer auf den Namen des Projekts im Projektmappen-Explorer von Visual Studio.

Da wir im Folgenden auch Änderungen an Datensätzen vornehmen wollen, soll bei jedem neuen Debuggen die Original-Datenbank erneut in das **Debug**-Verzeichnis kopiert werden. Dazu stellen Sie die Eigenschaft In Ausgabeverzeichnis kopieren des Elements **Suedsturm.mdb** auf den Wert **Immer kopieren** ein (siehe Bild 1). So erscheinen gelöschte oder geänderte beim nächsten Debuggen immer wieder im Ausgangszustand.

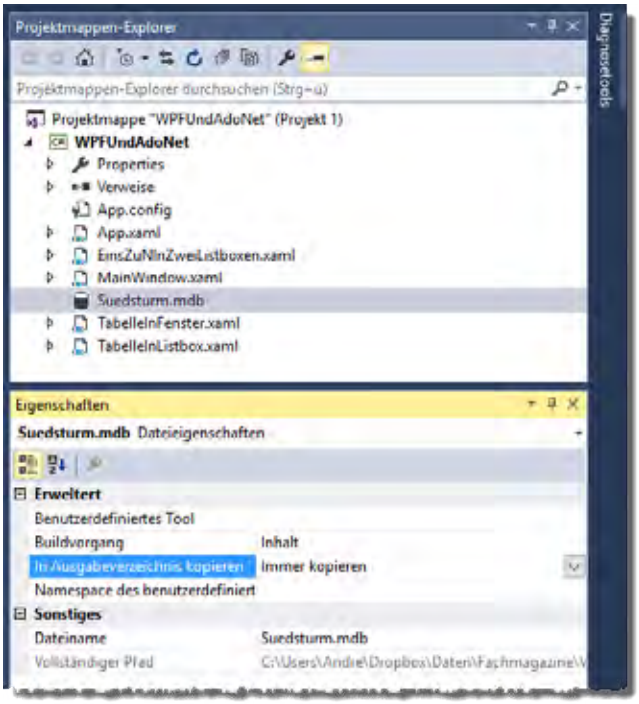


Bild 1: Die Datenbank soll beim Debuggen jeweils erneut ins Debug-Verzeichnis kopiert werden.

Schnelle Datenanzeige

Die einfachste Disziplin beim Anzeigen von Daten unter Access war ein Formular, das die Daten in der Formularansicht anzeigt. Dazu mussten Sie einfach nur die Tabelle markieren, deren Daten Sie anzeigen wollten, und den entsprechenden Assistenten aufrufen. Im Ergebnis erhielten Sie ein neues Formular mit allen Feldern der Datenherkunft plus einer Navigationsleiste, die fest im Formular verbaut war und die

Navigation zwischen den Datensätzen sowie das Anlegen und Löschen erleichterte.

Unter WPF, soviel sei vorab verraten, gibt es keine vorgefertigten Formulare mit Navigationsleisten oder gar einem Datensatzmarkierer wie unter Access. Hier müssen Sie alle Elemente selbst hinzufügen, allerdings hilft die Entwicklungsumgebung bei dem einen oder anderen Schritt.

Datenquelle hinzufügen

Mit dem Menüeintrag **Projekt|Neue Datenquelle hinzufügen...** öffnen Sie den Dialog **Assistent zum Konfigurieren von Datenquellen**. Wählen Sie hier im ersten Schritt den Eintrag **Datenbank** aus (siehe Bild 2).

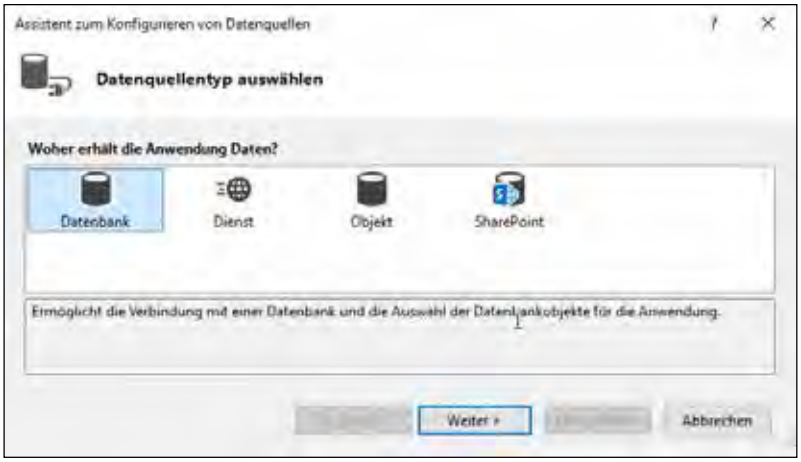


Bild 2: Der erste Schritt im **Assistent zum Konfigurieren von Datenquellen**

Im folgenden Schritt haben Sie nur eine sehr eingeschränkte Auswahl (siehe Bild 3). Sie brauchen also nur auf die Schaltfläche **Weiter** zu klicken.

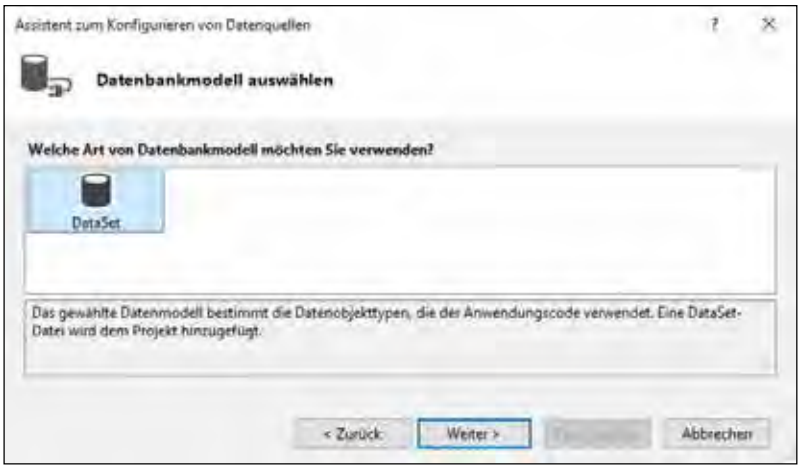


Bild 3: Auswahl der Art des Datenbankmodells

Im folgenden Dialog wählen Sie dann die Datenverbindung aus. In unserem Fall ist dies einfach: Wir müssen einfach nur die bereits als mögliche Datenquelle erkannte Access-Datenbank namens **Suedsturm.mdb** zu bestätigen. Mit einem Klick auf das Plus-Zeichen im unteren Bereich können Sie noch die Verbindungszeichenfolge für diese Verbindung anzeigen lassen (siehe Bild 4).

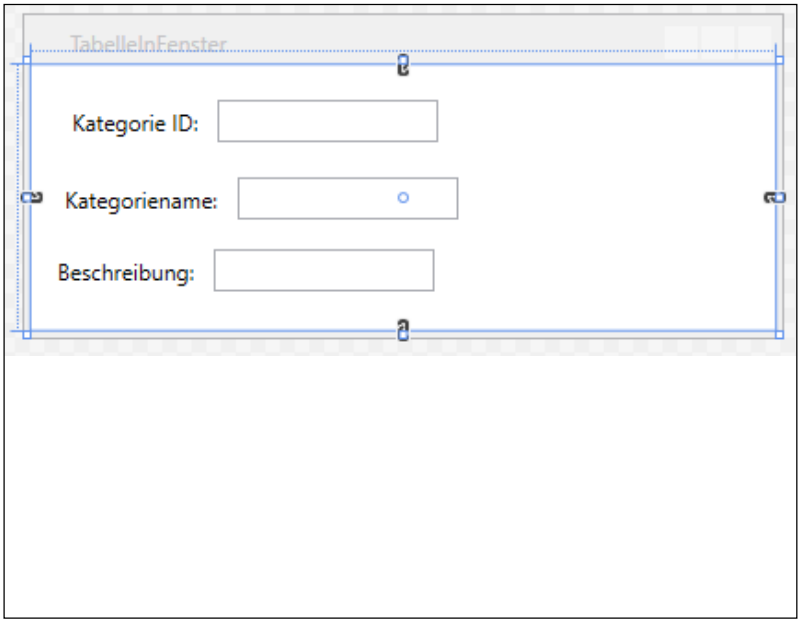


Bild 4: Festlegen der Datenverbindung

Schließlich können Sie noch festlegen, dass die Verbindungszeichenfolge in der Anwendungskonfigurationsdatei gespeichert wird (siehe Bild 5).

Im folgenden Schritt wird es interessant. Hier analysiert der Assistent die angegebene Datenbank und stellt alle Objekte der Typen **Tabelle**

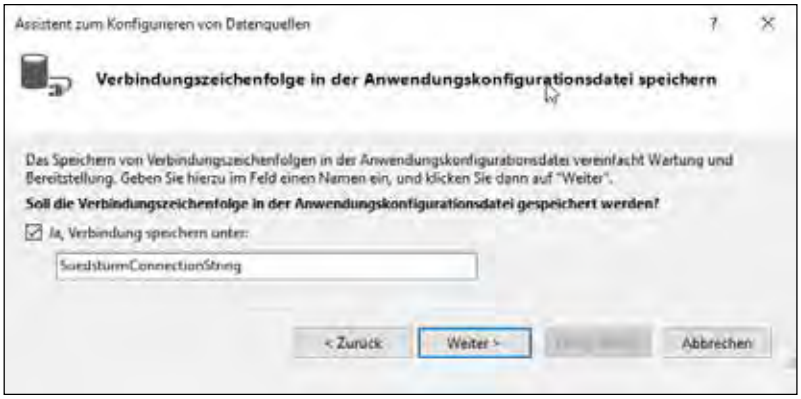


Bild 5: Speichern der Datenverbindung

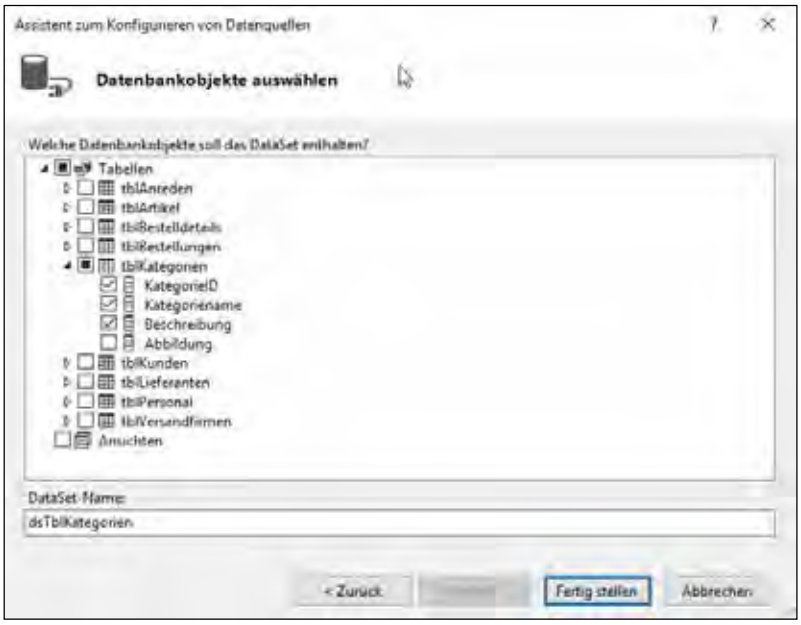


Bild 6: Auswahl der Tabellen/Datenbankobjekte für ein **DataSet**

und **Abfrage** zur Auswahl. Zu jedem Objekt können Sie die Felder auswählen, die zum **DataSet** hinzugefügt werden sollen. Überhaupt kommt hier das erste Mal der Begriff **DataSet** zum Einsatz (siehe Bild 6).

Fürs Erste wollen wir nun die drei Felder **KategorieID**, **Kategoriename** und **Beschreibung** der Tabelle **tblKategorien** zum **DataSet** hinzufügen. Dieses **DataSet** wollen wir dann unter dem Namen **dsTblKategorien** speichern.

Was ist nun geschehen? Der Assistent hat nicht nur einen neuen Eintrag zum Bereich **Datenquellen** hinzugefügt, sondern Sie finden auch im Projektmappen-Explorer ein neues Element. Dieses heißt **dsTblKategorien.xsd** und wird in Visual Studio grafisch wie in Bild 7 dargestellt. Dort finden Sie oben ein Objekt für die einzige Tabelle des **DataSet**-Objekts sowie ein weiteres namens **tblKategorienTableAdapter**, das die beiden Methoden **Fill** und **GetData()** anzeigt. Die untergeordneten Objekte enthalten, kurz gesagt, eine Menge automatisch generierten C#-Codes.

Das Ergebnis der Auswahl der Verbindungszeichenfolge landet übrigens in der Datei **Settings.settings** (siehe Bild 8). Wenn Sie die Verbindungszeichenfolge also entfernen möchten, müssen Sie also hier Hand anlegen.

Nach Abschluss des Assistenten finden Sie außerdem die neue Datenquelle im Bereich **Datenquellen** von Visual Studio vor (siehe Bild 9).

Felder zum Fenster hinzufügen

Mit den erzeugten Elementen können wir nun schon eine Menge erledigen. Der erste Schritt ist, dass Sie ein neues Fenster namens **TabelleInFenster.xaml** anlegen (Menüpunkt **Projekt|Fenster hinzufügen**). Das Fenster ist nun noch leer. Nun aktivieren Sie den Bereich **Ansicht|Weitere Fenster|Datenquellen** beziehungsweise Tastenkombination **Umschalt + Alt + D** (siehe Bild 10).

Hier haben Sie eine Reihe von Möglichkeiten. Die erste ist, einfach das Objekt **tblKategorien** in das Fenster zu ziehen. Dies würde ein **DataGrid**-Steuerelement zum Fenster hinzufügen, das die drei Felder der Tabelle **tblKategorien** anzeigt. Dies behandeln wir allerdings in einem weiteren Artikel – hier wollen wir uns um die Detailansicht kümmern.

Dazu ziehen Sie die drei Felder **KategorieID**, **Kategoriename** und **Beschreibung** einzeln in den Entwurf des Fensters,

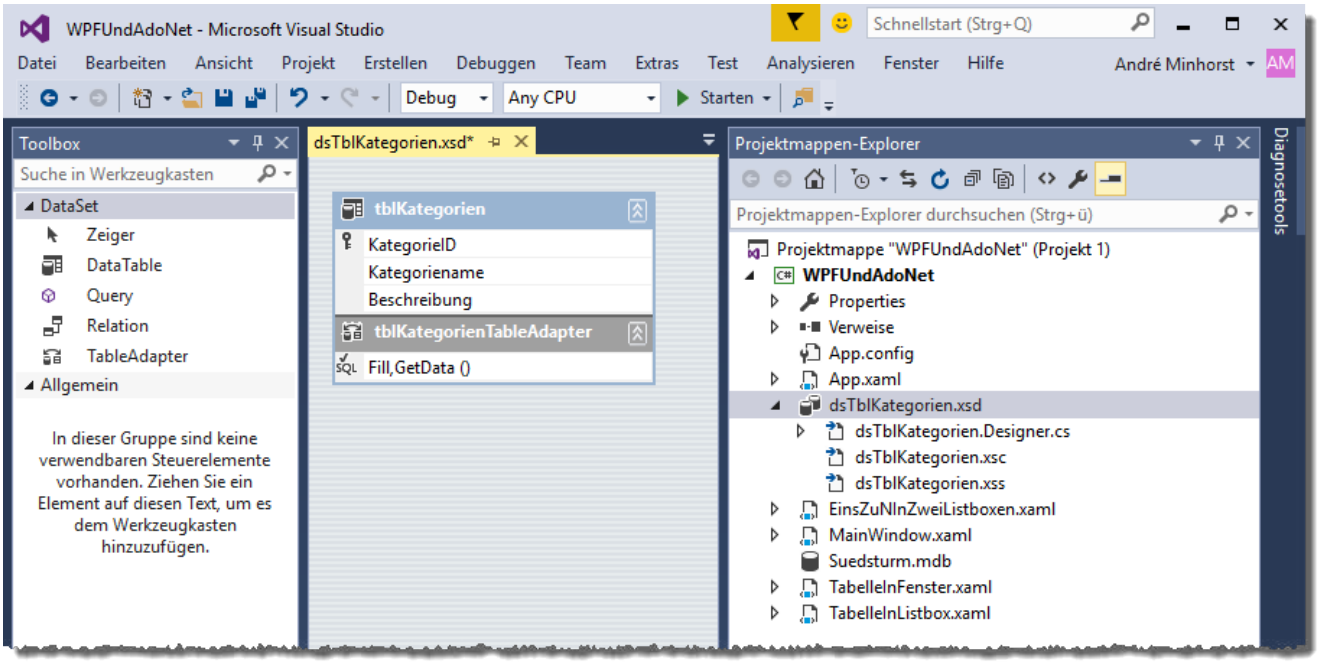


Bild 7: Neue Objekte im Projektmappen-Explorer

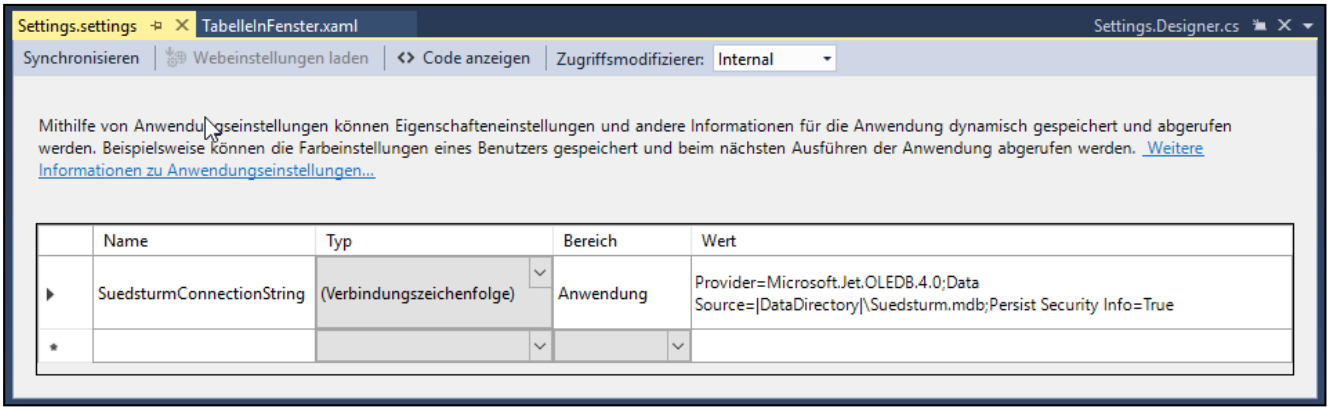


Bild 8: Die Verbindungszeichenfolge wird im Bereich **Settings.settings** gespeichert.

sodass sein Inhalt etwa wie in Bild 11 aussieht. Was ist nun geschehen?

- Visual Studio hat dem **.xaml**-Code eine Verbindung zum soeben erstellen und gespeicherten DataSet **dsTblKategorien** sowie eine **CollectionViewSource**, deren Funktion wir weiter unten erläutern.

- Das **Window**-Element wurde um das Ereignisattribut **Loaded** erweitert: **<Window ... Loaded="Window_Loaded">**. Passend dazu wurde in der Datei **TabelleInFenster.xaml.cs** eine entsprechende Methode angelegt, die beim Eintreten des Ereignisses ausgelöst wird.

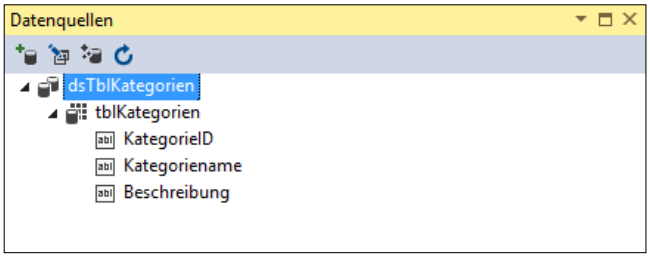


Bild 9: Die neue Datenquelle im Bereich **Datenquellen** von Visual Studio

- Außerdem wurden natürlich die drei **TextBox**-Steuerelemente plus entsprechender **Label**-Elemente angelegt – allerdings jedes in einem eigenen **Grid**-Konstrukt, die keine vernünftige Ausrichtung erlaubt. Dies passen wir noch an.

Steuerelemente organisieren

Schauen wir uns erst die Steuerelemente und die **Grid**-Elemente an, in denen diese angelegt wurden. Alle wurden mit dem Attribut **DataContext="{StaticResource tblKategorienViewSource}"** ausgestattet. Es reicht, wenn wir dieses einmal dem umschließenden **Grid**-Element zuweisen.

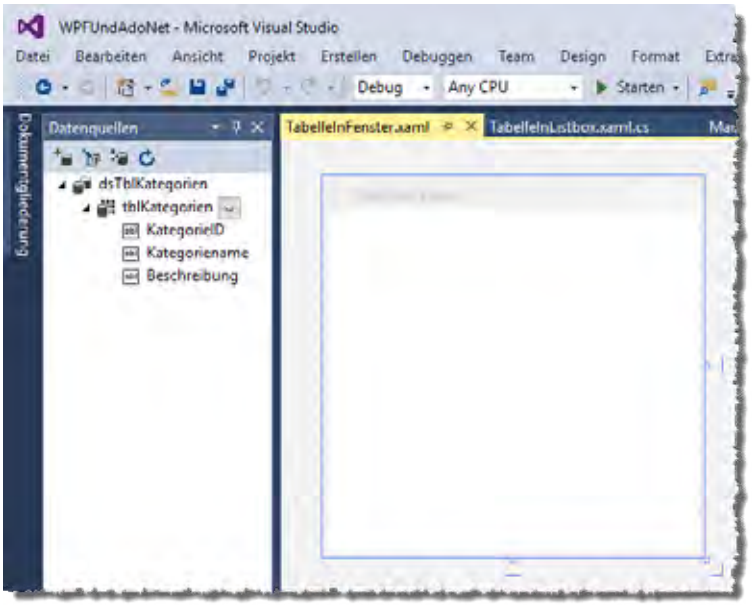


Bild 10: Die Datenquellen und das noch leere Fenster

Dann stellen wir das übergeordnete **Grid**-Steuerelement auf drei Zeilen und zwei Spalten ein. Die übrigen **Grid**-Elemente samt **Grid.ColumnDefinitions**-, **ColumnDefinition**-, **Grid.RowDefinitions**- und **RowDefinition**-Elementen löschen wir.

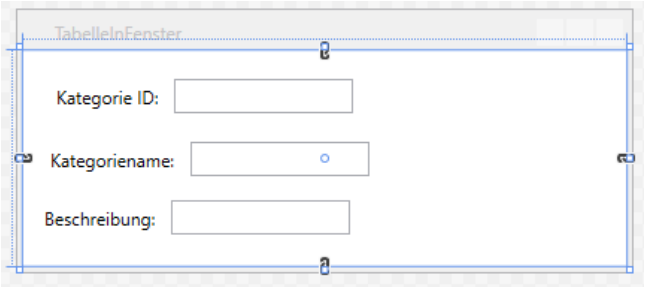


Bild 11: Fenster mit Steuerelementen

Die verbleibenden **Label**- und **TextBox**-Steuerelemente fügen wir in die **Grid**-Zellen ein, indem wir die Attribute **Grid.Column** und **Grid.Row** entsprechend mit den nullbasierten Indizes der Zellen auszeichnen. Außerdem passen wir das Layout noch an. Das Ergebnis im Code sehen Sie in Listing 2.

Datensatz im Fenster anzeigen

Wenn Sie nun das Debugging starten (**F5**), erhalten Sie das Fenster aus Bild 12. Es funktioniert! Das Fenster zeigt den ersten Datensatz der festgelegten Datenquelle an. Das ging nicht so schnell wie mit Access und wir haben ja auch noch keine Navigationsschaltflächen hinzugefügt, aber immerhin sehen wir schon einmal Daten aus der Datenbank.

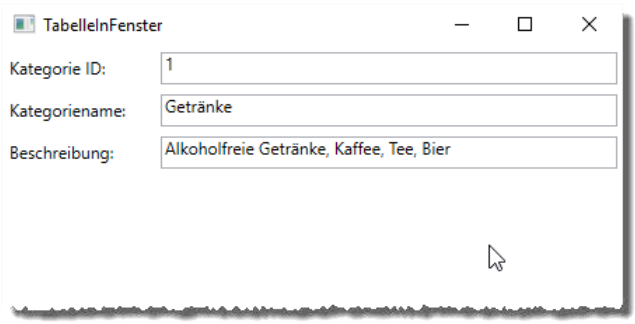


Bild 12: Tabelleninhalte im WPF-Fenster

```
<Window.Resources>
    <WPFUndAdoNet:dsTblKategorien x:Key="dsTblKategorien"/>
    <CollectionViewSource x:Key="tblKategorienViewSource"
        Source="{Binding tblKategorien, Source={StaticResource dsTblKategorien}}"/>
</Window.Resources>
```

Listing 1: Ressourcen-Informationen durch Hinzufügen des ersten Steuerelements zum Fenster

```
<Window xmlns:WPFUndAdoNet="clr-namespace:WPFUndAdoNet" x:Class="WPFUndAdoNet.Properties.TabelleInFenster"
        mc:Ignorable="d" Title="TabelleInFenster" Height="175" Width="412" Loaded="Window_Loaded">
    <Window.Resources>
        <WPFUndAdoNet:dsTblKategorien x:Key="dsTblKategorien"/>
        <CollectionViewSource x:Key="tblKategorienViewSource" Source="{Binding tblKategorien,
            Source={StaticResource dsTblKategorien}}"/>
    </Window.Resources>
    <Grid DataContext="{StaticResource tblKategorienViewSource}">
        <Grid.RowDefinitions>
            <RowDefinition Height="30"/>
            <RowDefinition Height="30"/>
            <RowDefinition Height="30"/>
        </Grid.RowDefinitions>
        <Grid.ColumnDefinitions>
            <ColumnDefinition Width="110"/>
            <ColumnDefinition/>
        </Grid.ColumnDefinitions>
        <Label Content="Kategorie ID:" Grid.Column="0" HorizontalAlignment="Left" Grid.Row="0"
            VerticalAlignment="Center"/>
        <TextBox x:Name="kategorieIDTextBox" Grid.Column="1" HorizontalAlignment="Stretch" Height="23" Margin="3,3,3,3"
            Grid.Row="0" Text="{Binding KategorieID, Mode=TwoWay, NotifyOnValidationError=true,
            ValidatesOnExceptions=true}" VerticalAlignment="Center" />
        <Label Content="Kategoriename:" Grid.Column="0" HorizontalAlignment="Left" Grid.Row="1"
            VerticalAlignment="Center"/>
        <TextBox x:Name="kategorienameTextBox" Grid.Column="1" HorizontalAlignment="Stretch" Height="23" Margin="3,3,3,3"
            Grid.Row="1" Text="{Binding Kategoriename, Mode=TwoWay, NotifyOnValidationError=true,
            ValidatesOnExceptions=true}" VerticalAlignment="Center"/>
        <Label Content="Beschreibung:" Grid.Column="0" HorizontalAlignment="Left" Grid.Row="2"
            VerticalAlignment="Center"/>
        <TextBox x:Name="beschreibungTextBox" Grid.Column="1" HorizontalAlignment="Stretch" Height="23" Margin="3,3,3,3"
            Grid.Row="2" Text="{Binding Beschreibung, Mode=TwoWay, NotifyOnValidationError=true,
            ValidatesOnExceptions=true}" VerticalAlignment="Center"/>
    </Grid>
</Window>
```

Listing 2: Code des WPF-Fensters

Navigationsschaltflächen hinzufügen

Damit stellen wir uns natürlich nicht zufrieden, denn wer zeigt schon immer den ersten Datensatz in einem Fenster an? Wir benötigen Schaltflächen, mit denen Sie zum ersten und zum letzten Datensatz navigieren können sowie zum vorherigen und zum folgenden Datensatz. Und damit kommen wir auch zu dem bereits von Visual Studio beim Hineinziehen des ersten Feldes aus der Datenquelle angelegten Code in der Datei **TabellenInFenster.xaml.cs**.

Wenn Sie sich dort die Methode **Window_Loaded** ansehen, finden Sie einige Anweisungen, die – um einige führende Angaben erleichtert – wie in Listing 3 aussehen.

Die erste Anweisung erzeugt ein Objekt auf Basis der Klasse **dsTblKategorien**, die in der beim Erstellen der Datenquelle erzeugten Klasse **dsTblKategorien.Designer.cs** steckt. Die Klasse erbt von der Klasse **DataSet** des Namespaces **Windows.Data** und wird hier genau wie ein **DataSet** verwendet.

Methodenstarter als Vorlage

In Ausgabe 2/2015 haben wir unter dem Titel »**Experimentieren mit der Konsole**« eine Technik vorgestellt, mit der Sie die Methoden einer Klasse direkt in der Konsole auflisten und dort zur Ausführung auswählen können. Der Nachteil war, dass Sie damit nur eine Klasse mit fest vorgegebenem Klassennamen referenzieren konnten. Dies haben wir nun erweitert: mit einer neuen »Main«-Methode, die beim Start der Anwendung aufgerufen wird und alle Methoden aller Klassen außer der Klasse »Program« selbst auflistet und so ermöglicht, diese zu starten.

Im vorliegenden Beitrag nun stellen wir Ihnen eine neue Version der Technik aus Experimentieren mit der Konsole vor, mit der Sie die zu testenden Methoden bequem in anderen Klassen anlegen, die sich optimalerweise auch noch in einer anderen Datei befinden. Damit wird das Ausprobieren von Code über einfache C#-Methoden noch einfacher als zuvor.

Alles, was Sie tun müssen, ist das Erstellen eines neuen Projekts als Konsolen-Projekt, dessen Datei **Program.cs** Sie mit dem Code aus Listing 1 füllen.

Listing 2 liefert den Rest des notwendigen Programmcodes. Anschließend brauchen Sie nur noch ein neues Klassenmodul anzulegen und dort die zu testende Methode einzufügen – beispielsweise die folgende:

```
[Beschreibung("Dies ist die erste Testmethode.")]
public static void Test1()
{
    Console.WriteLine("Test erfolgreich!");
    Console.ReadLine();
}
```

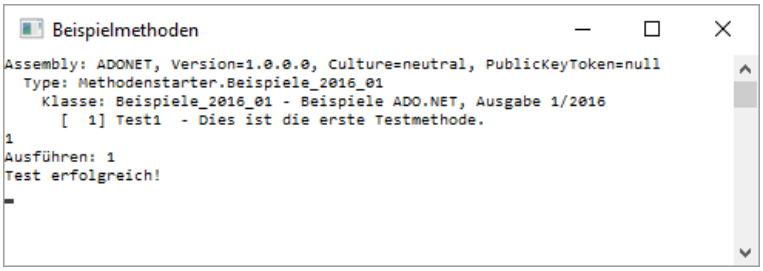


Bild 1: Anzeige der Beispielmethoden einer Klasse

Wenn Sie in Ihrem neuen Klassenmodul den gleichen Namespace verwenden, in dem sich auch die Klasse **Program** mit der Methode **Main** befindet (in diesem Beispiel also den Namespace namens **Methodenstarter**), sind Sie nun bereits fertig und können das Projekt zum Debuggen starten. Das Ergebnis sieht dann wie in Bild 1 aus.

Sollten Sie im neuen Klassenmodul hingegen einen neuen Namespace verwenden, müssen Sie noch den Namespace mit der **Main**-Methode etwa per **using**-Direktive bekanntgeben:

```
using Methodenstarter;
```

Schnelles Testprojekt erstellen

Nun möchten Sie vielleicht gelegentlich auf die Schnelle ein neues Projekt erstellen, um mal eben ein paar C#-Befehle zu testen oder auch um die Beispiele aus einem Artikel aus dem **DATENBANKENTWICKLER** auszuprobieren.

In diesem Fall haben Sie die hier vorgestellten Methoden vielleicht gerade nicht greifbar. Damit Sie immer auf die Methode **Main** zugreifen können, legen wir einfach eine Vorlage an, die genau dies liefert. Die neue Vorlage finden Sie dann im **Neues Projekt**-Fenster.

Aber wie erstellen Sie eine Projektvorlage? Das ist gar nicht so schwer: Als Erstes müssen Sie das Projekt erstellen, das alle Elemente für die gewünschte Vorlage enthält, und dieses Projekt dann in eine Vorlage umwandeln. Dazu wählen Sie einfach den

```
using System;
using System.Reflection;
using System.Collections;

namespace Methodenstarter {
    class Program {
        static void Main(string[] args) {
            int j = 0; int i = 1; int intType = 1;
            string strMethode = ""; string strBeschreibung = "";
            ArrayList Methoden = new ArrayList(); ArrayList Typen = new ArrayList();
            Type[] types; Type currentType = null;
            do {
                i = 1;
                Assembly assembly = Assembly.GetEntryAssembly();
                Console.WriteLine("Assembly: {0}", assembly.FullName);
                types = FindAllExportedTypesInNamespaceFor(assembly);
                Array.ForEach(types, delegate (Type type) {
                    if ((type.Name!="Program") & (type.Name!="Beschreibung")) {
                        intType = ++intType;
                        currentType = type;
                        Console.WriteLine("  Type: {0}", type.FullName);
                        Beschreibung beschreibung = (Beschreibung)type.GetCustomAttribute(typeof(Beschreibung));
                        if (beschreibung != null) { strBeschreibung = " - " + beschreibung.Text; }
                        else { strBeschreibung = ""; }
                        Console.WriteLine("    Klasse: {0}{1}", type.Name, strBeschreibung);
                        MethodInfo[] myArrayMethodInfo = type.GetMethods(BindingFlags.Public | BindingFlags.Static |
                            BindingFlags.Instance | BindingFlags.DeclaredOnly);
                        strMethode = "";
                        foreach (MethodInfo methodInfo in myArrayMethodInfo) {
                            beschreibung = (Beschreibung)methodInfo.GetCustomAttribute(typeof(Beschreibung));
                            if (beschreibung != null) { strBeschreibung = " - " + beschreibung.Text; }
                            else { strBeschreibung = ""; }
                            strMethode = methodInfo.Name;
                            Console.WriteLine("      [{0,3}] {1} {2}", i++, strMethode, strBeschreibung);
                            Methoden.Add(strMethode);
                            Typen.Add(type);
                        }
```

Die Leseprobe dieses Artikels ist hier zu Ende.

Willst Du mehr?

ZUM SHOP

2.400 Seiten Know-how rund um die Datenbankentwicklung mit .NET

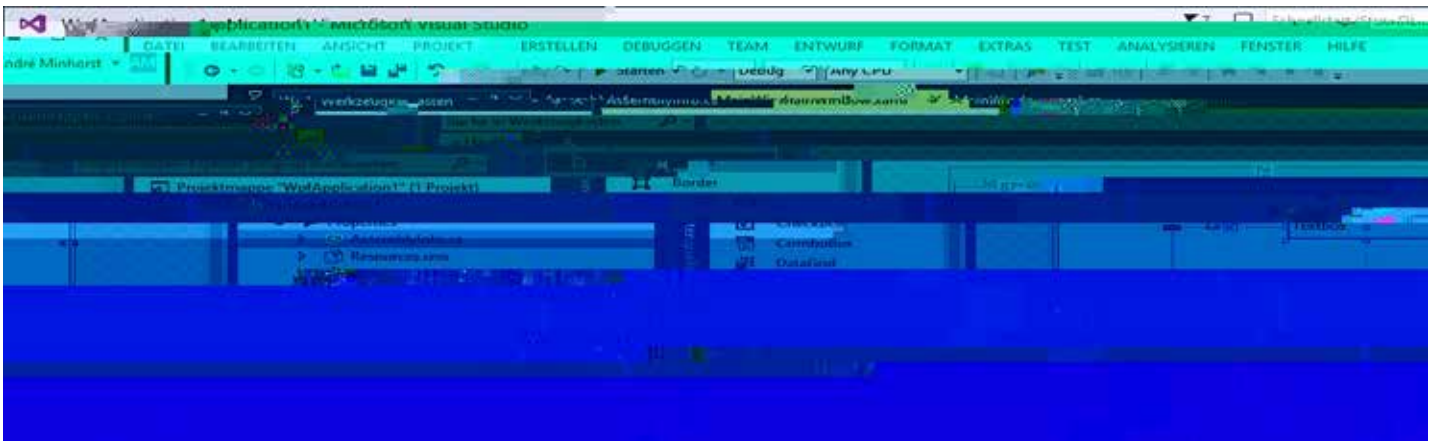
André Minhorst Verlag

Spare 20 EUR mit Code dbew20

AUSGABE 02/2016

DATENBANK ENTWICKLER

MAGAZIN FÜR DIE DATENBANKENTWICKLUNG MIT VISUAL STUDIO FÜR DESKTOP, WEB UND CO.



TOP-THEMEN:

WPF-BASICS	Steuerelemente binden	SEITE 3
WPF-CONTROLS	Schaltflächen	SEITE 10
VON ACCESS ZU WPF	Fenster mit 1:n-Daten und Lookup-Feld	SEITE 27
VON ACCESS ZU WPF	Kombinationsfelder mit Daten füllen	SEITE 34
DATENZUGRIFF	SQL Server installieren	SEITE 40

André Minhorst Verlag

Bindung zwischen Steuerelementen

Unter VBA waren die Möglichkeiten der Datenbindung überschaubar: Formulare, Berichte und einige Steuerelemente wiesen entsprechende Eigenschaften auf, die an Tabellen, Abfragen oder auch Wertlisten oder Felder geknüpft werden konnten. Unter WPF und C# sieht es ganz anders aus. Hier stehen Objekte im Vordergrund. Dieser Artikel zeigt daher zunächst, wie Sie Elemente wie Steuerelemente an die Eigenschaften anderer Elemente binden.

Einführung

Wenn Sie zuvor mit Access programmiert haben und dieses Magazin lesen, um zu erfahren, wie das, was Sie zuvor mit Access erledigt haben, mit C#/WPF gelingt, wollen Sie vermutlich nur eins: Daten aus Tabellen oder Abfragen in Fenstern und Steuerelementen anzeigen und diese zum Bearbeiten bereitstellen. Beim Thema Datenbindung ist WPF/C# allerdings etwas breiter aufgestellt, da es nicht nur auf Datenquellen wie Tabellen oder Abfragen fixiert ist, sondern primär auf Objekte als Datenlieferant.

Daher räumen wir das Feld etwas weiter auf und beginnen nicht gleich damit, die Daten per Fenster und Steuerelementen bereitzustellen. Zunächst einmal schauen wir uns an, wie Sie verschiedene Elemente an einander binden können. Davon abgesehen haben Sie ja in vorhergehenden Artikeln der Kategorie **Datenzugriffstechnik** bereits einige Beispiele erhalten, wie dies gelingt. Grundsätzlich dient die Bindung aber dazu, nicht nur die Eigenschaften eines Elements an ein anderes Element zu binden, sondern die gebundenen Daten können auch aus Quellen wie einer XML-Datei, Auflistungen oder Datenbanken stammen.

Von Textfeld zu Textfeld

Im ersten Beispiel wollen wir schlicht und einfach, dass ein Textfeld den Inhalt eines anderen Textfeldes anzeigt. Unter Access/VBA haben Sie dazu das **Nach Aktualisierung**-Ereignis genutzt – oder auch das **Bei Änderung**-Ereignis. Das Erste hat beispielsweise auf die Eingabe- oder Tabulatortaste reagiert, Letzteres auf jede einzelne Änderung des Inhalts. Etwas eleganter konnten Sie dies erledigen, wenn Sie das erste Textfeld gleich der Eigenschaft **Steuerelementinhalt** des zweiten Textfeldes zugewiesen haben. Damit war noch nicht einmal mehr Code notwendig, um den Inhalt des ersten Textfeldes nach Änderungen in das zweite Textfeld zu übertragen.

Unter WPF/C# wollen wir uns gleich um die zweite Variante kümmern, also das zweite Textfeld an das erste Textfeld binden. Dazu fügen Sie einem Fenster eines neuen Projekts zwei Textfelder namens **txtFeld1** und **txtFeld2** hinzu:

```
<Window x:Class="Datenbindung.MainWindow" ...
Title="MainWindow" Height="350" Width="525">
    <Grid>
        <Label x:Name="lbl1" ... />
        <Label x:Name="lbl2" ... />
        <TextBox x:Name="txtFeld1" ... />
        <TextBox x:Name="txtFeld2" ... />
    </Grid>
</Window>
```

Damit können Sie nun den Inhalt der beiden Textfelder beliebig ändern – es tut sich nichts. Dies ändern wir, indem wir die Methode **MainWindow** im C#-Modul des Fensters wie folgt um vier Zeilen erweitern:

```
public MainWindow() {
    InitializeComponent();
    Binding binding = new Binding();
    binding.ElementName = "txtFeld1";
    binding.Path = new PropertyPath("Text");
    txtFeld2.SetBinding(TextBox.TextProperty, binding);
}
```

Diese vier Zeilen binden das zweite Feld so an das erste Feld, dass dieses immer den Inhalt des ersten Feldes anzeigt (siehe Bild 1). In diesem Fall werden Änderungen am Text des ersten Textfeldes direkt in das zweite Textfeld übertragen. Änderungen am zweiten Textfeld landen erst nach dem Verlassen im ersten Textfeld – wie Sie dieses Verhalten ändern, erfahren Sie weiter unten.

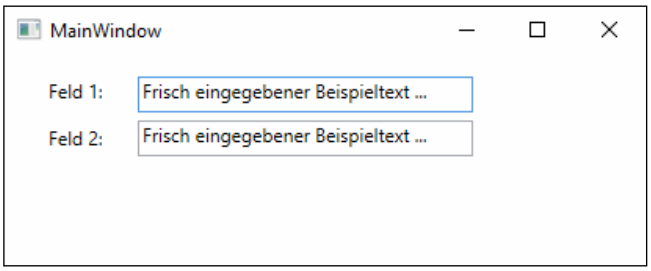


Bild 1: Das untere Textfeld ist an das obere Textfeld gebunden.

Schauen wir uns die vier Zeilen im Detail an. Die erste Zeile erzeugt ein neues Objekt des Typs **Binding**:

```
Binding binding = new Binding();
```

Die zweite Zeile weist der Eigenschaft **ElementName** des **Binding**-Objekts den Namen des ersten Textfelds **txtFeld1** zu:

```
binding.ElementName = "txtFeld1";
```

Im Gegensatz zu Access/VBA, wo sich der Steuerelementinhalt eines Steuerelements immer auf den Steuerelementinhalt des angegebenen Quellsteuerelements bezogen hat, können Sie unter WPF alle Eigenschaften des mit Source angegebenen Objekts als Quellwert nutzen. In diesem Fall ist dies die Eigenschaft **Text**, die wir als neues **PropertyPath**-Objekt der Eigenschaft **Path** des **Binding**-Objekts zuweisen:

```
binding.Path = new PropertyPath("Text");
```

Schließlich müssen wir noch festlegen, welches Steuerelement das neue **Binding**-Objekt nutzen soll und welches die Zieleigenschaft ist. Dies erledigen wir mit der Methode **SetBinding** wie in der folgenden Anweisung. Der erste Parameter ist die Zieleigenschaft, hier **TextProperty**, der zweite das **Binding**-Objekt:

```
txtFeld2.SetBinding(TextBox.TextProperty, binding);
```

Die Binding-Klasse und die Dependency Property

Die hier verwendete **Binding**-Klasse legt also fest, wie die Bindung zwischen einer Komponente wie etwa einem Steuerelement und der Datenquelle, hier die Eigenschaft eines Steuerelements, erfolgt. Dabei haben wir hier bereits die folgenden Eigenschaften genutzt:

- **ElementName**: Name des als Datenquelle verwendeten Steuerelements
- **Path**: Eigenschaft, aus der die Daten stammen

Mit dem ersten Parameter der **SetBinding**-Methode haben wir oberflächlich betrachtet einfach die Eigenschaft des Objekts **txtFeld2** festgelegt, das gebunden werden soll. Aber warum geben wir dann nicht den Namen der Eigenschaft, also **Text**, sondern **TextBox.TextProperty** als Parameter an? Weil dieser Parameter eine so genannte **Dependency Property**, zu Deutsch Abhängigkeitseigenschaft, erwartet.

Diese Dependency Properties haben die in Zusammenhang mit dem Binding interessante Eigenschaft, Automatismen für die Benachrichtigung über Änderungen bereitzustellen – in diesem Fall für den Zweck, Änderungen am Quellwert an das Ziel zu übergeben und nach Wunsch auch umgekehrt. Für Access-Nutzer ist das in etwa mit der Synchronisierung von Haupt- und Unterformular über die beiden Eigenschaften **Verknüpfen von** und **Verknüpfen nach** vergleichbar – trägt man dort die Feldnamen der verknüpften Felder der beiden Tabellen aus Haupt- und Unterformular ein, zeigt das Unterformular automatisch die zum aktuellen Datensatz im Hauptformular passenden Datensätze des Unterformulars an.

Weitere Eigenschaften der Binding-Klasse

Die **Binding**-Klasse liefert noch weitere interessante Eigenschaften:

- **Converter**: Gibt einen möglichen Konverter für die zu bindenden Daten an.

- **FallbackValue**: Wert, den das Ziel annehmen soll, falls das gebundene Objekt oder die Eigenschaft den Wert **Null** liefert.

- **IsAsync**: Legt mit **True** fest, dass die Bindung asynchron hergestellt wird.

- **Mode**: Legt fest, wie die Bindung hergestellt wird. Mögliche Werte: **Default** (übernimmt den Standardwert für die jeweilige Dependency Property), **OneTime** (nur einmalige Aktualisierung), **OneWay** (ändert nur von der Quelle zum Ziel, aber nicht umgekehrt), **OneWayToSource** (ändert nur vom Ziel zur Quelle), **TwoWay** (ändert von der Quelle zum Ziel und umgekehrt)

- **Source**: Quellobjekt der Datenbindung

- **TargetNullValue**: Wert, falls das gebundene Element keinen Wert liefert, also nicht vorhanden ist (siehe weiter unten)

- **UpdateSourceTrigger**: Gibt an, wie die Quelle in Abhängigkeit vom Inhalt des Ziels geändert werden soll. In unserem Beispiel ändert sich der Inhalt der Quelle, sobald das geänderte Zieltextfeld den Fokus verliert. Dies entspricht der Einstellung **LostFocus**. Wenn Sie den Wert **PropertyChanged** verwenden, wird auch die Quelle nach der Eingabe eines jeden Zeichens in das Zieltextfeld geändert (siehe Beispiel 5).

Textfelder binden per XAML

Das Ganze lässt sich per XAML noch einfacher abbilden. Hierzu fügen wir dem Formular zwei weitere Textfelder namens **txtFeld3** und **txtFeld4** hinzu. Für das zweite ändern Sie das Attribut **Text**, das ja zuvor eine Zeichenkette enthielt, wie folgt (siehe Beispiel 2 im Beispielprojekt):

```
<TextBox x:Name="txtFeld4" ...
    Text="{Binding ElementName=txtFeld3, Path=Text}"/>
```

Was bedeutet **{Binding ElementName=txtFeld3, Path=Text}** nun? **Binding** entspricht dem Objekt des Typs **Binding**, das wir oben per Code mit der **New**-Anweisung erzeugen mussten. Dahinter geben wir den Namen des Quell-Elements an, also **txtFeld3**. Schließlich folgt der Pfad zur zu bindenden Eigenschaft, hier also **Text** (bei einer ComboBox könnte dies beispielsweise auch **SelectedItem.Content** lauten). Es gibt noch eine alternative Schreibweise:

```
<TextBox x:Name="txtFeld4" ...
    Text="{Binding Text, ElementName=txtFeld3}" />
```

Hier geben wir mit **Text** gleich den Namen der Eigenschaft des Quellobjekts an, an das wir die Eigenschaft **Text** des Zielobjekts binden wollen. Das Quell-Element weisen wir schließlich über das Name-Wert-Paar **ElementName=txtFeld3** zu. Der Pfad zur Eigenschaft muss hier zwingend als erstes Argument hinter dem Schlüsselwort **Binding** angegeben werden.

Da wir das Binding der Eigenschaft **Text** des Textfeldes **txtFeld4** zugewiesen haben, haben wir alle Informationen zusammen, die wir weiter oben in vier Zeilen C#-Code programmiert haben. Das Ergebnis überzeugt, denn die Textfelder werden wie die C#-Version synchronisiert, wenn Sie den Inhalt von **txtFeld3** ändern. Eine übersichtlichere Schreibweise hierfür ohne geschweifte Klammern sieht so aus (siehe Beispiel 3 im Beispielprojekt):

```
<TextBox x:Name="txtFeld4" ...>
    <TextBox.Text>
        <Binding ElementName="txtFeld3" Path="Text" />
    </TextBox.Text>
</TextBox>
```

Standardwerte bei Bindungen

Es gibt zwei Attribute, mit denen wir eine Art Standardwert simulieren können. Warum gleich zwei? Weil es bei einer Bindung verschiedene Fälle gibt, die dafür sorgen, dass kein

WPF-Controls: Schaltflächen

In den ersten Ausgaben des DATENBANKENTWICKLERs haben Sie bereits einige Steuerelemente kurz kennen gelernt. In den folgenden Ausgaben wird sich dies ändern. Im vorliegenden Artikel schauen wir uns die verschiedenen Schaltflächen an, die WPF uns zur Verfügung stellt. Dabei lernen Sie die wichtigsten Unterschiede zu den von Access bekannten Schaltflächen kennen.

Allgemeines zu WPF-Steuerelementen

WPF-Fenster und auch die enthaltenen Steuerelemente wie zum Beispiel die Schaltflächen passen sich standardmäßig an das jeweils ausgewählte Windows-Design an. Dieses stellen Sie in der Systemsteuerung unter **Systemsteuerung|Darstellung und Anpassung|Anpassung** ein. In Bild 1 sehen Sie beispielsweise ein Fenster mit einer einfachen Schaltfläche unter drei verschiedenen Windows-Designs. Davon abgesehen bietet WPF unendliche Möglichkeiten zum Gestalten der Benutzeroberfläche – vor allem, wenn Sie zuvor mit Access gearbeitet haben. Die Gestaltungsmöglichkeiten unter Access haben sich zwar mit den letzten Versionen leicht verbessert, aber mit den Möglichkeiten von WPF ist dies nicht vergleichbar. Die Gestaltung spielt jedoch aktuell eine untergeordnete Rolle, sodass wir viel später darauf eingehen werden.

Alle Steuerelemente?

Die Toolbox, die zusammen mit dem Entwurf eines WPF-Fensters eingeblendet wird, zeigt im oberen Bereich die häufig verwendeten WPF-Steuerelemente an, im unteren Bereich können Sie dann alle WPF-Steuerelemente einblenden (siehe Bild 2).

Aber sind dies nun wirklich alle Steuerelemente? Nein: Es gibt noch einige weitere Steuerelemente wie beispielsweise den **RepeatButton** oder den **ToggleButton**, die in der Liste gar nicht erscheinen. Wie aber fügen Sie diese zu einem WPF-Fenster hinzu? Ganz einfach: Sie legen das Steuerelement über den XAML-Editor an. Geben Sie das Kleiner-Zeichen (**<**) gefolgt vom ersten Buchstaben des gesuchten

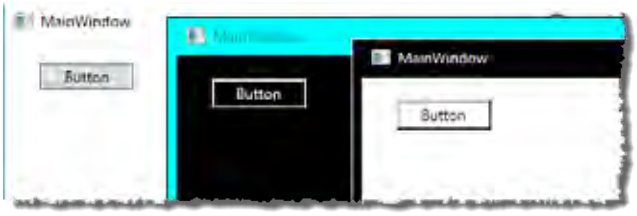


Bild 1: WPF-Fenster mit einer Schaltfläche in drei verschiedenen Designs

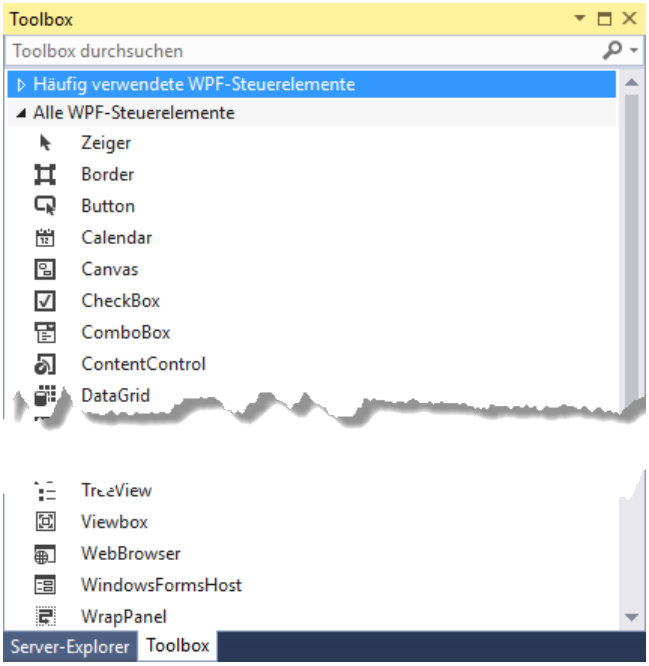


Bild 2: Toolbox mit »allen« WPF-Steuerelementen

Elements ein, erhalten Sie direkt eine Liste aller in Frage kommenden Steuerelemente (siehe Bild 3).

Das Button-Steuerelement

Wenn Sie von Access kommen, wollen Sie sicher erstmal das Pendant zur dortigen Schaltfläche kennen lernen. Dabei han-

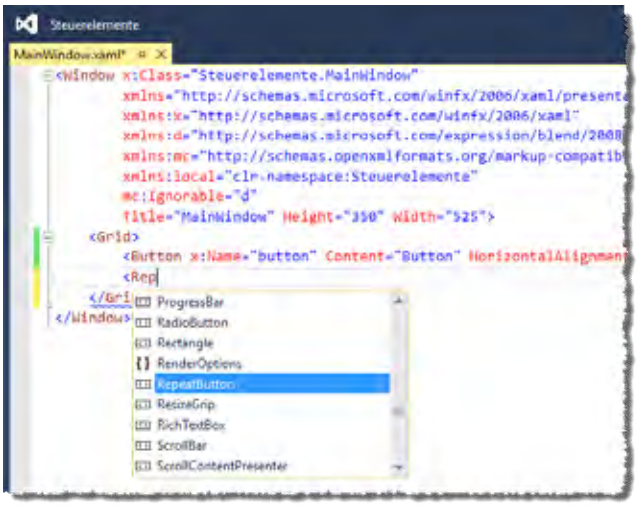


Bild 3: Anlegen nicht in der Toolbox verfügbarer Steuerelemente

delt es sich um das Button-Element, dass Sie sowohl über die Toolbox als auch direkt per XAML hinzufügen können. Da Sie per XAML immer alle Attribute manuell setzen müssen, macht es Sinn, dass Button-Element aus der Toolbox in das Fenster zu ziehen und dann seine Attribute im XAML-Editor anzupassen (oder auch per Eigenschaftsfenster, wenn Sie mögen – wir wollen hier allerdings beim XAML-Editor bleiben). Das Steuerelement wird standardmäßig mit Werten für die Attribute **Name**, **Content**, **HorizontalAlignment**, **Margin**, **VerticalAlignment** und **Width** ausgestattet. **Name** enthält den Steuerelementnamen, **Content** den angezeigten Text.

Entspricht **Content** also der Eigenschaft **Text** der Schaltfläche von Access? Mitnichten: Die Anzeige eines Textes ist nur die einfachste Variante, die Schaltfläche zu füllen. Sie können dort auch wesentlich komplexere Inhalte anzeigen – beispielsweise Icon und Text nebeneinander.

Beim Klicken

Als Nächstes schauen wir uns an, wie die WPF-Schaltfläche ein Ereignis auslöst. Unter Access haben wir die Ereigniseigenschaft **Beim Klicken** mit dem Wert **[Ereignisprozedur]** gefüllt und per Klick die passende Ereignisprozedur angelegt, die wir nur noch mit den gewünschten Anweisungen füllen mussten. Hier läuft das ganz ähnlich. Sie fügen dem **Button**-Element das Attribut **click** hinzu.

Spätestens nach Eingabe des Gleichheitszeichens erscheint ein kleines Popup und bietet den Wert **<Neuer Ereignishandler>** zur Auswahl an (siehe Bild 4).

Klicken Sie darauf, fügt Visual Studio den Wert **<Buttonname>_Click** als Wert des Attributs ein und legt automatisch die folgende Methode an, wobei **<Buttonname>** als Platzhalter für den aktuellen Namen des Buttons zu verstehen ist:

```
private void <Buttonname>_Click(object sender,
                                RoutedEventArgs e) {
    ...
}
```

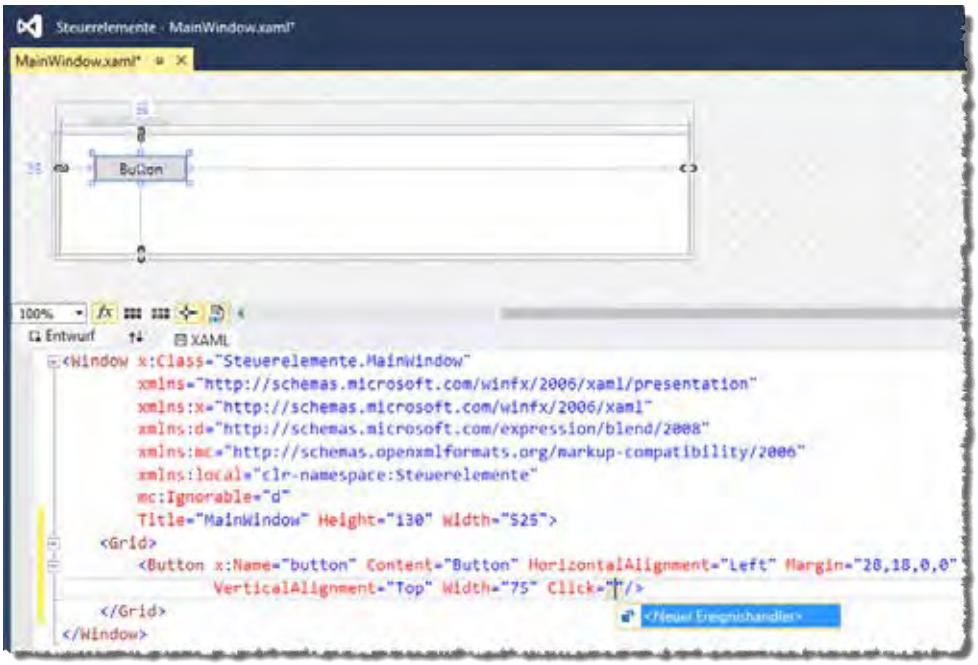


Bild 4: Hinzufügen des **Click**-Ereignisses

WPF-Controls: Kombinationsfelder

In den ersten Ausgaben des DATENBANKENTWICKLERs haben Sie bereits einige Steuerelemente kurz kennen gelernt. In den folgenden Ausgaben wird sich dies ändern. Im vorliegenden Artikel schauen wir uns das ComboBox-Steuerelement an, dass Sie als Access-Entwickler ja vielleicht aus dem Eff-Eff kennen. Unter WPF läuft aber gewöhnlich alles etwas anders, sodass wir hier mit den Grundlagen einsteigen und uns dann Schritt für Schritt Themen wir der Datenbindung et cetera widmen. Außerdem dürfen wir vornewegnehmen, dass sicher hier einige Möglichkeiten mehr ergeben ...

Wenn Sie ein **ComboBox**-Steuerelement zu einem Fenster hinzufügen und das Debugging starten, erhalten Sie ein einfaches Kombinationsfeld, das beim Aufklappen keine Werte anzeigt und in das Sie auch keinen Text einfügen können. Das ist der erste Unterschied im Vergleich zu einem Access-Kombinationsfeld – dort können Sie zumindest gleich einen Wert eintippen.

Eine recht einfache Definition eines solchen **ComboBox**-Steuerelements sieht so aus:

```
<ComboBox x:Name="comboBox"
HorizontalAlignment="Left" Margin="127,10,0,0"
VerticalAlignment="Top" Width="120"/>
```

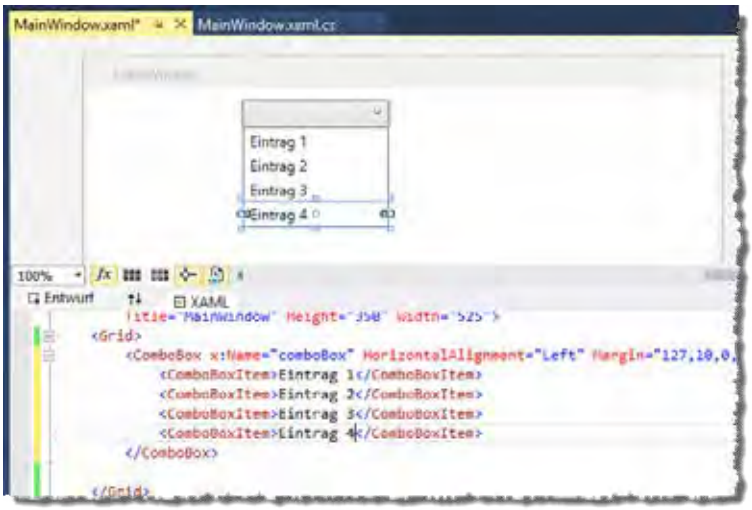


Bild 1: WPF-Fenster mit einer Schaltfläche in drei verschiedenen Designs

ComboBox mit statischen Werten

Soll die **ComboBox** beim Öffnen des Fensters immer die gleichen Werte anzeigen, können Sie diese ganz einfach durch eine entsprechende XAML-Definition hinzufügen. Dies ähnelt dann dem Anlegen von Einträgen in einem HTML-Select-Element, allerdings verwenden Sie für jeden neuen Eintrag ein **ComboBoxItem**-Element, das Sie unterhalt des **ComboBox**-Elements anlegen:

```
<ComboBox x:Name="comboBox" ... >
    <ComboBoxItem>Eintrag 1</ComboBoxItem>
    <ComboBoxItem>Eintrag 2</ComboBoxItem>
    <ComboBoxItem>Eintrag 3</ComboBoxItem>
    <ComboBoxItem>Eintrag 4</ComboBoxItem>
</ComboBox>
```

Während Sie diese Einträge unterhalb des **ComboBox**-Elements hinzufügen, zeigt der XAML-Editor direkt die Einträge in der Vorschau an (siehe Bild 1).

Im Betrieb erscheint dieses Kombinationsfeld nun wie in Bild 2. Beim Aufklappen zeigt es die vier Einträge an, die Auswahl eines Eintrags trägt diesen oben in das Steuerelement ein. Nun wollen wir dies noch erweitern, und zwar erstens durch die Eingabe von Texten durch den Benutzer.

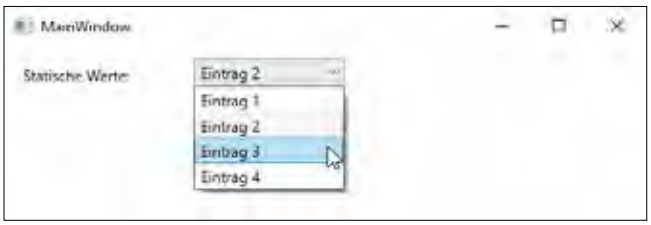


Bild 2: Statisches Kombinationsfeld mit vier Einträgen

ComboBox bedienen

Natürlich wissen Sie, dass Sie eine ComboBox mit einem Mausklick auf den Pfeil nach unten öffnen und dass Sie dort Text eingeben und Einträge auswählen können. Aber haben Sie auch schon einmal nur mit der Tastatur auf eine ComboBox zugegriffen? Sie können diese nämlich auch durch Betätigen der Taste **F4** öffnen oder schließen. Auch die Tastenkombinationen **Alt + Nach oben** oder **Alt + Nach unten** öffnen und schließen die Liste des **ComboBox**-Steuerelements. Schließlich blättern Sie auch bei nicht ausgeklappter Liste mit den Tasten **Nach oben** und **Nach unten** durch die Einträge, wobei Sie **Nach oben** vom obersten Element und **Nach unten** vom untersten Element aus keine Auswirkungen zeigen.

Eingabe von Werten aktivieren

Um das Eingabefeld des Kombinationsfeldes zu aktivieren, fügen Sie einfach nur das Attribut **IsEditable** zum **ComboBox**-Element hinzu und stellen es auf den Wert **True** ein:

```
<ComboBox x:Name="cboMitEingabe" ... IsEditable="True">
    <ComboBoxItem Content="Eintrag 1"/>
    ...
</ComboBox>
```

Damit können Sie nun Werte wie in Bild 3 eingeben. Allerdings fügt dies den Wert noch nicht zur Auswahlliste hinzu.

Wenn Sie den Anfangsbuchstaben eines der bereits in der Liste enthaltenen Einträge eingeben, wird dieser übrigens gleich als Eintrag vorgeschlagen (siehe Bild 4) – wie auch unter Access. Nach der Eingabe eines Wertes in das Textfeld prüft die ComboBox, ob dieser Wert bereits in der Liste enthalten ist. Falls ja, wird dieser Eintrag als markiert gekennzeichnet – später schauen wir uns an, wie wir dies nutzen

Bild 4: Autoergänzung standardmäßig

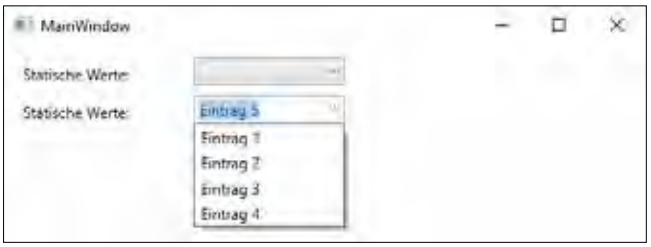


Bild 3: Hinzufügen eines eigenen Textes

können. Ist der Wert nicht vorhanden, hat die Liste keinen selektierten Eintrag. In beiden Fällen wird der Wert jedoch in die Eigenschaft **Text** des **ComboBox**-Steuerelements eingetragen.

IsEditable und IsReadOnly

Selbst, wenn Sie das Attribut **IsEditable** auf **True** eingestellt haben, können Sie die Eingabe von Texten in das Textfeld des Kombinationsfeldes unterbinden. Dazu stellen Sie die Eigenschaft **IsReadOnly** ebenfalls auf **True** ein. Der Unterschied zwischen **IsEditable=False** sowie **IsEditable=True** und **IsReadOnly=True** ist, dass der Hintergrund des Eingabefeldes nicht grau, sondern weiß dargestellt wird.

```
<ComboBox x:Name="cboMitEingabe" ... IsEditable="True"
        IsReadOnly="True">
    ...
</ComboBox>
```

Ein Element vorauswählen per XAML

Unter Access/VBA haben Sie etwa die folgende Zeile verwendet, um beispielsweise den ersten Eintrag eines Kombinationsfeldes etwa beim Öffnen des Formulars zu markieren:

```
Me!cboEintraege = Me!cboEintraege.ItemData(0)
```

Unter WPF können Sie dieses Element gleich in der Definition markieren, sofern diese überhaupt per Definition festgelegt und nicht dynamisch mit einer anderen Datenherkunft gefüllt wird. Dazu fügen Sie dem zu markierenden Eintrag das Attribut **IsSelected** mit dem Wert **True** hinzu:

```
<ComboBox x:Name="cboMitStandardwert" ...
        IsEditable="True" IsReadOnly="True">
```

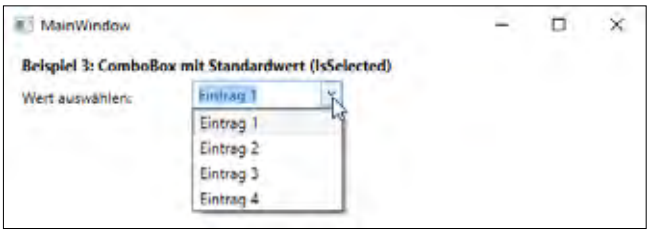


Bild 5: ComboBox mit voreingestelltem Wert

```
    <ComboBoxItem Content="Eintrag 1" IsSelected="True"/>
    <ComboBoxItem Content="Eintrag 2"/>
    <ComboBoxItem Content="Eintrag 3"/>
    <ComboBoxItem Content="Eintrag 4"/>
</ComboBox>
```

Das Ergebnis finden Sie in Bild 5. Der Text des gewählten Eintrags wird nicht nur im Eingabefeld angezeigt, sondern beim Ausklappen der Liste auch optisch hervorgehoben (siehe Beispiel 3a des Beispielprojekts).

Ein Element vorauswählen per C#

Natürlich können Sie ein Element auch per Code auswählen wie unter Access/VBA. Dazu nutzen wir die Ereignismethode, die durch das Ereignis **Loaded** des Fensters ausgelöst wird. Damit diese bekannt ist, fügen wir die Eigenschaft **Loaded** zum **Window**-Element hinzu (dazu den Attributnamen eingeben, Tabulator-Taste drücken und den dann erscheinenden Eintrag **<Neuer Ereignishandler>** auswählen, um die Methode gleich automatisch anzulegen):

```
<Window x:Class="ComboBoxBeispiele.MainWindow" ...
Title="MainWindow" ... Loaded="Window_Loaded">
```

Diese Methode füllen Sie im C#-Modul zum Fenster wie folgt mit einer einzigen Anweisung auf:

```
private void Window_Loaded(object sender, RoutedEventArgs e) {
    cboMitStandardwertCSharp.SelectedIndex = 0;
}
```

Diese stellt den Index des aktivierten Eintrags des **ComboBox**-Steuerelements auf den Wert **0** ein, was dem ersten Eintrag entspricht (siehe Beispiel 3b des Beispielprojekts)

Vorauswahl mit Loaded-Ereignis der ComboBox

Das **ComboBox**-Steuerelement besitzt selbst ein **Loaded**-Attribut. Dieses wird beim Laden des Steuerelements ausgelöst. Um die **Loaded**-Methode des **Window**-Elements nicht zu überfrachten, können Sie auch die des **ComboBox**-Elements nutzen, um dessen Standardwert zu definieren:

```
<ComboBox x:Name="cboMitStandardwertLoaded" ...
    Loaded="cboMitStandardwertLoaded_Loaded">
```

Die **Loaded**-Methode enthält prinzipiell die gleiche Anweisung

Ereignisse beim Auf- und Zuklappen

Im Gegensatz zu Access, wo Sie zum Beispiel auf die Auswahl eines Elements reagieren konnten, gibt es bei der WPF-ComboBox zwei Ereignisse, die durch das Auf- und Zuklappen der Liste ausgelöst werden:

- **DropDownOpened**: Wird beim Aufklappen der Liste ausgelöst.
- **DropDownClosed**: Wird beim Zuklappen der Liste ausgelöst.

Außerdem gibt es mit **IsDropDownOpen** noch eine Eigenschaft, mit der Sie auslesen können, ob die Liste gerade auf- oder zugeklappt ist. Um diese zu testen, fügen wir dem **ComboBox**-Element die Namen der Methoden zu, die beim Eintreten der Ereignisse **DropDownClosed** und **DropDownOpened** ausgeführt werden sollen. Das **ComboBox**-Element sieht dann so aus:

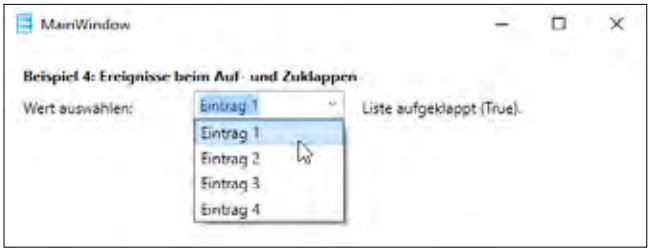


Bild 6: Auslösen von Ereignissen beim Auf- und Zuklappen und Ermitteln des Status

WPF-Controls: ToolTips

Unter Access/VBA fristeten die ToolTips ein Schattendasein. Sie konnten dafür lediglich einen Text angeben, der beim Überfahren des jeweiligen Steuerelements eingeblendet wurde und dann wieder verschwand. Es gab keinerlei Möglichkeiten, das Aussehen des ToolTips zu beeinflussen. Unter WPF sieht dies ganz anders aus: Sie können nicht nur das Aussehen und den Inhalt vielfältig gestalten, sondern auch noch bestimmen, wann und wie der ToolTip ein- und ausgeblendet wird.

ToolTip für Button-Steuerelemente

Von Access/VBA haben Sie mit der Eigenschaft **SteuerelementTip-Text** (unter VBA **ControlTipText**) festgelegt, welcher Text angezeigt werden soll, wenn der Benutzer mit dem Mauszeiger über dem Steuerelement verweilt. Eine solche Eigenschaft gibt es natürlich auch unter WPF. Am einfachsten fügen Sie einen solchen Hinweistext hinzu, indem Sie einfach das entsprechende Attribut des betroffenen Steuerelements verwenden – in diesem Fall für ein **Button**-Element:

```
<Button x:Name="btnToolTip" Content="Button mit ToolTip"
... ToolTip="Ich bin ein Tooltip für einen Button!"/>
```

Das Ergebnis sieht dann etwa wie in Bild 1 aus.

Das **ToolTip**-Element ist tatsächlich ein echtes Steuerelement, das Sie mit beliebigen Inhalten füllen können. Wie das aussehen kann, sehen Sie in Bild 2.

Der notwendige Code ist in Listing 1 abgebildet. Hier ist zu beachten, dass wir nicht mehr das Attribut **ToolTip** verwenden. Stattdessen fügen wir dem **Button**-Element ein schließendes **Button**-Element hinzu (**</Button>**) und fügen einige weitere Elemente innerhalb von **<Button>** und **</Button>** ein.

Dabei handelt es sich zunächst um das **Button.ToolTip**-Element, das ohne eigene Attribute kommt, sondern nur den Rahmen für die anzuzeigenden Inhalte liefert. Darin bringen wir ein **StackPanel**-Element unter. Dieses haben Sie bisher in diesem Magazin noch nicht kennen gelernt, daher in aller

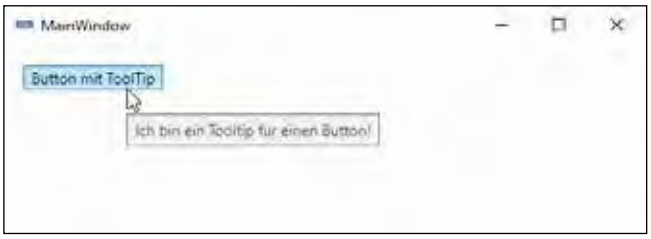


Bild 1: Schaltfläche mit einem einfachen **ToolTip**-Text

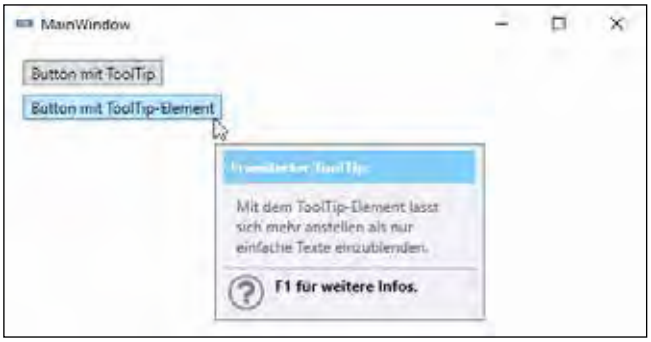


Bild 2: Erweiterter **ToolTip**-Text

Kürze: Es handelt sich dabei wie beim Grid um ein Element zur einfachen Anordnung von Inhalten in vertikaler oder horizontaler Ausrichtung.

Das erste **StackPanel**-Element enthält einige untereinander darzustellende Elemente: Ein **Label**, einen **TextBlock**, ein Linie (**Line**) und ein weiteres **StackPanel**. Dieses soll seine Unterelemente nun horizontal ausrichten, also nebeneinander – und zwar ein **Image**-Element und ein **Label**-Element.

Damit das dem **Image**-Element zugewiesene Bild angezeigt wird, müssen Sie dieses dem Projekt hinzufügen. Im Beispielprojekt ist dieses bereits enthalten.

```
<Button x:Name="btnToolTipElement" Content="Button mit ToolTip-Element" ...>
    <Button.ToolTip>
        <StackPanel>
            <Label Content="Erweiterter ToolTip" FontWeight="Bold" Background="LightSkyBlue" Foreground="White" />
            <TextBlock Padding="10" TextWrapping="WrapWithOverflow" Width="200">
               Mit dem ToolTip-Element lässt sich mehr anstellen als nur einfache Texte einzublenden.
            </TextBlock>
            <Line Stroke="Black" StrokeThickness="1" X2="200"/>
            <StackPanel Orientation="Horizontal">
                <Image Margin="2" Source="question_32.png" />
                <Label FontWeight="Bold" Content="F1 für weitere Infos." />
            </StackPanel>
        </StackPanel>
    </Button.ToolTip>
</Button>
```

Listing 1: Einen ausgefeilten **ToolTip** zu einem **Button**-Element hinzufügen

Zeitvorgaben für Einblenden, Ausblenden et cetera

Sie können mit drei Eigenschaften das Verhalten festlegen, wie schnell ToolTip-Texte mit weiteren Informationen ein- und ausgeblendet werden.

Diese sind Eigenschaften des Objekts **ToolTipService**:

- **InitialShowDelay**: Gibt die Anzahl der Millisekunden an, bis der ToolTip nach Positionieren des Mauszeigers auf dem Steuerelement eingeblendet wird.
- **ShowDuration**: Gibt die Anzahl der Millisekunden an, bis der ToolTip wieder ausgeblendet wird.
- **BetweenShowDelay**: Betrifft Fenster mit mehreren ToolTips. Wenn Sie für ein Steuerelement **ToolTipService.BetweenShowDelay** etwa auf **3.000** einstellen, dann wird innerhalb dieser drei Sekunden jeder **ToolTip** anderer Steuerelemente unmittelbar nach dem Überfahren mit der Maus angezeigt – egal, welchen Wert die Eigenschaft **InitialShowDelay** aufweist.

Die folgenden drei Schaltflächen zeigen, wie Sie die zeitbezogenen Attribute festlegen. Beachten Sie, dass die Attribute nicht direkt als Attribute des übergeordneten Elements, sondern als Attribute des Elements **ToolTipService** angegeben werden:

```
<Button x:Name="btnVerzoegerung" Content="Verzögerung beim Anzeigen" ... ToolTip="ToolTip mit verzögerter Anzeige (drei Sekunden)" ToolTipService.InitialShowDelay="3000" />

<Button x:Name="btnAusblenden" Content="Ausblenden nach 5 Sekunden" ... ToolTip="ToolTip mit schnellem Ausblenden (eine Sekunde)" ToolTipService.ShowDuration="1000"/>

<Button x:Name="btnBetweenShowDelay" Content="Andere schnell anzeigen" ... ToolTip="Innerhalb von drei Sekunden werden ToolTips, auch wenn InitialShowDelay größer ist, direkt angezeigt." ToolTipService.BetweenShowDelay="3000"/>
```

Platzierung der ToolTip-Elemente

Normalerweise erscheint der ToolTip immer dort, wo sich der Mauszeiger gerade befindet. Das Objekt **ToolTipService** liefert jedoch einige Eigenschaften, mit denen Sie die Position des ToolTips sehr genau festlegen können. Sie sind dabei noch nicht einmal auf eine Position abhängig vom Mauszeiger beschränkt, sondern können auch eine absolute Position vom linken oberen Bildschirmrand oder vom Steuerelement aus festlegen.

Fenster mit 1:n-Daten und Lookup-Feld

Im vorherigen Artikel der Kategorie »Von Access zu WPF« haben wir Daten einer einfachen Tabelle in einem Fenster abgebildet. Nun gehen wir einen Schritt weiter und schauen uns die Abbildung zweier per 1:n-Beziehung verknüpfter Tabellen an, wobei die Daten der einen Tabelle per Kombinationsfeld als Lookup-Wert ausgewählt werden sollen. Eines der bekanntesten Beispiele für eine solche Lookup-Tabelle ist die Tabelle tblAnreden. Den anderen Teil übernimmt selbstverständlich die Tabelle tblKunden.

Fenster erstellen

Wir wollen also ein Fenster erstellen, dass die Daten der Tabelle **tblKunden** anzeigt und zur Auswahl einer Anrede für den jeweils angezeigten Kunden ein Kombinationsfeld zur Auswahl eines Datensatzes der Tabelle **tblAnreden** zur Verfügung stellt.

Dazu fügen wir einem Projekt auf Basis der Vorlage **Visual C#IWPF-Vorlagen** zunächst ein neues Fenster namens **TabelleMitLookupDaten** hinzu.

Datenbank hinzufügen

Wir wollen, wie in anderen Beispielen auch, wieder mit der Access-Datenbank **Suedsturm.mdb** arbeiten. Fügen Sie diese daher zum Projekt hinzu. Das gelingt am einfachsten, indem Sie die Datenbank aus dem Windows Explorer einfach auf den Namen des Projekts im Projektmappen-Explorer ziehen.

DataSet hinzufügen

Wie im Beispiel zur Anzeige der Daten einer einfachen Tabelle wollen wir auch hier wieder ein typisiertes DataSet erstellen, das heißt: Wir lassen uns von Visual Studio eine Menge Arbeit abnehmen, indem wir einen Assistenten nutzen.

Um diesen zu starten, wählen Sie den Menüeintrag **Projektl Neue Datenquelle hinzufügen**.

Im nun erscheinenden Assistent zum Konfigurieren von Datenquellen klicken Sie im ersten Schritt unter der Überschrift **Datenquellentyp auswählen** auf den Eintrag **Datenbank**.

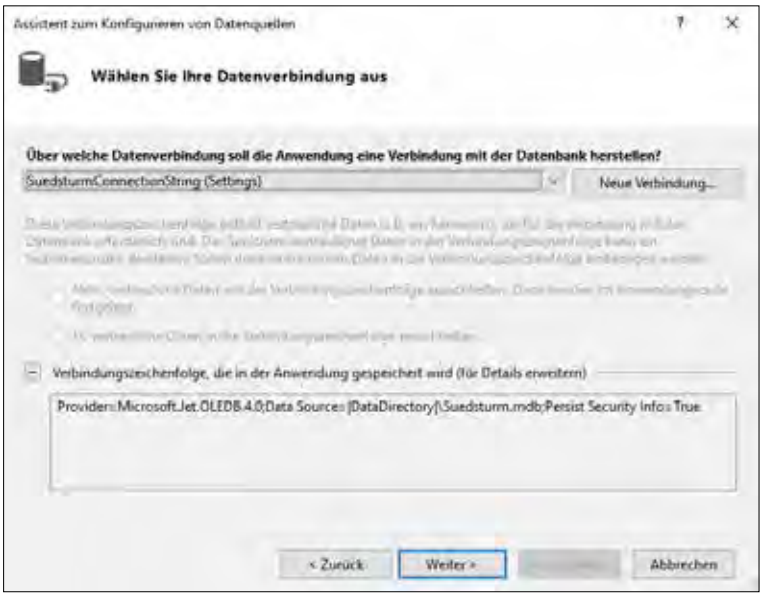


Bild 1: Auswahl einer vorhandenen Datenverbindung

Damit gelangen wir schnell zum zweiten Schritt namens **Datenbankmodell auswählen**. Hier gibt es nicht viel Auswahl, nämlich nur den Eintrag **DataSet**, den Sie folgerichtig ebenfalls doppelt anklicken, um auch diesen Schritt schnell zu beenden.

Sollten Sie das neue Beispiel in der Beispieldatenbank aus dem Artikel **Fenster mit einfachen Tabellendaten** anlegen, sieht der folgende Dialogschritt wie in aus: Es liegt nämlich bereits eine Datenverbindung namens **SuedsturmConnectionString** vor, die Sie beibehalten können (siehe Bild 1). Wenn Sie mit einem neuen, leeren Projekt begonnen haben – kein Problem! Sofern Sie, wie oben angegeben, die Datenbank **Suedsturm.mdb** zum Projekt hinzugefügt haben,

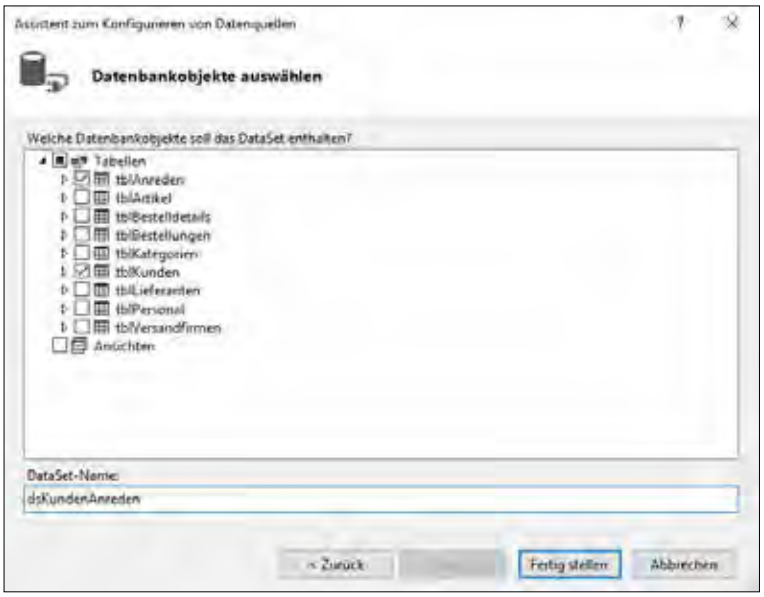


Bild 2: Auswahl der Tabellen und Angabe des Namens für das DataSet

erscheint diese als Auswahlmöglichkeit im genannten Dialog. Klicken Sie hier auf **Weiter**, fragt der Assistent Sie noch, ob Sie die Verbindung unter dem Namen **SuedsturmConnectionString** speichern wollen, was Sie akzeptieren. Beim Anlegen des nächsten DataSets können Sie dann auch auf den Eintrag **SuedsturmConnectionString** zugreifen.

Tabellen für das DataSet zusammenstellen

Nun wollen wir die Tabellen festlegen, die wir im DataSet benötigen. Anders als unter Access benötigen wir hier nicht nur die die Tabelle **tblKunden**, die dort, ein entsprechendes Nachschlagefeld vorausgesetzt, gleich die Daten der Tabelle **tblAnreden** mitliefert, sondern fügen beide Tabellen zum DataSet hinzu. Nach der Auswahl der Tabellen **tblKunden** und **tblAnreden** geben Sie als Namen für das DataSet den Wert **dsKundenAnreden** an (siehe Bild 2).

Damit legt Visual Studio wieder ein Element namens **dsKundenAnreden.xsd** im Projektmappen-Explorer an, dass die Definition des typisierten DataSets und auch die entsprechenden Objekte, Eigenschaften und Methoden bereithält.

Daten der Tabelle tblKunden zum Fenster hinzufügen

Nun fügen wir die Felder der Tabelle **tblKunden** zum Fenster **TabelleMitLookupdaten** hinzu.

Dazu aktivieren Sie zunächst das Datenquellen-Fenster (**Alt + Umschalt + D** oder **Ansicht|Fenster|Datenquellen**). Dieses zeigt nun (gegebenenfalls neben anderen) das DataSet **dsKundenAnreden** mit den beiden untergeordneten Tabellen **tblKunden** und **tblAnreden** an. Wir wollen ein Detailformular für die Darstellung eines Kunden anlegen, also bereiten wir den Eintrag **tblKunden** im **Datenquellen**-Fenster nun darauf vor. Dazu klicken Sie auf **tblKunden**, sodass sich der Eintrag in ein Kombinationsfeld ändert und öffnen dieses. Hier legen Sie den Wert **Details** fest (siehe Bild 3).

Der Eintrag **tblKunden** hat nun ein neues Icon erhalten, was auf die geänderte Ansicht beim Hinzufügen in ein Fenster hinweist (siehe Bild 4). Wenn Sie die Tabelle nun in das Fenster ziehen, erscheinen bereits die gewünschten Felder

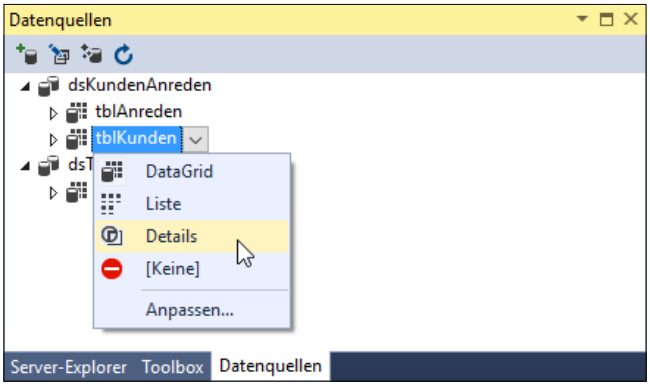


Bild 3: Einstellen der Darstellung der Felder der Tabelle **tblKunden**

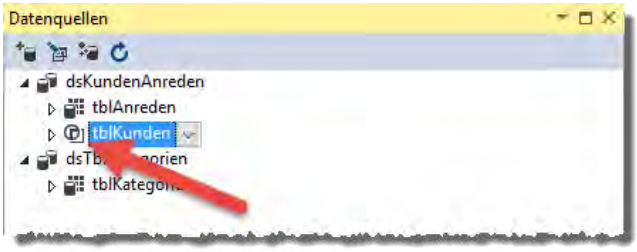


Bild 4: **tblKunden** mit neuem Ansichtstyp

samt Bezeichnungsfeldern – allerdings alle als **TextBox**-Steuerelement. Wir möchten aber doch ein Kombinationsfeld zur Auswahl der Anreden anlegen!

Doch auch dafür sieht der **Datenquellen**-Dialog eine Möglichkeit vor. Klappen Sie dort den Eintrag **tblKunden** auf und klicken Sie dann auf **AnredeID**. Auch hier erscheint ein Kombinationsfeld, aus dem Sie den Eintrag **ComboBox** auswählen (siehe Bild 5).

Damit haben Sie den Steuerelement-Typ für dieses Feld geändert, wie auch die anschließende Anzeige eines neuen Icons für den Eintrag bestätigt (siehe Bild 6).

Wenn Sie nun die Tabelle **tblKunden** in das WPF-Fenster ziehen, erscheint möglicherweise die Fehlermeldung aus Bild 7. In diesem Fall erstellen Sie das Projekt einfach einmal mit dem Menüeintrag **Erstellen|<Projektname> erstellen** (**<Projektname>** durch den aktuellen Projektnamen ersetzen). Im nächsten Anlauf sollte es dann gelingen.

Ziehen Sie dann die Tabelle **tblKunden** in die linke, obere Ecke des WPF-Fenster wie in Bild 8 dargestellt. Die Darstellung im Entwurf sieht zunächst etwas missglückt aus, was aber daran liegt, dass die Inhalte noch nicht initialisiert wurden.

Sie können das Projekt beispielsweise einmal starten oder auch nur eine Eigenschaft des WPF-Fenster aktualisieren (zum Beispiel die Höhe), um die Inhalt der **Label**- und **TextBox**-Steuerelemente korrekt darzustellen.

> Das Beispielprojekt startet das Fenster **MainWindow**, von dem Sie die einzelnen WPF-Beispiele per Mausklick öffnen können.

Nach dem Starten der Anwendung sieht das WPF-Fenster schon recht gut aus – allein beim Feld **AnredeID** gibt es Probleme: Es zeigt nur die gespeicherten Werte an, also die Primärschlüsselwerte der Tabelle **tblAnreden**. Allerdings nicht nur **1** und **2**, sondern beide Werte mehrfach und in

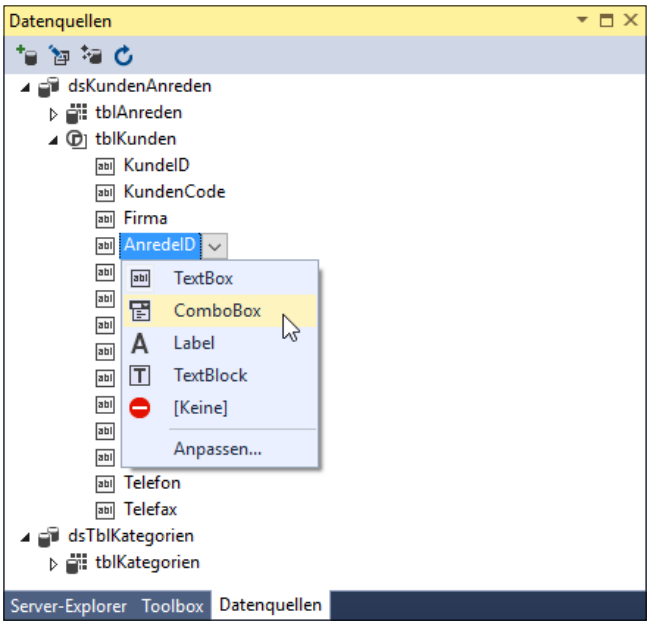


Bild 5: Ändern des Typs des Steuerelements für das Feld **AnredeID**

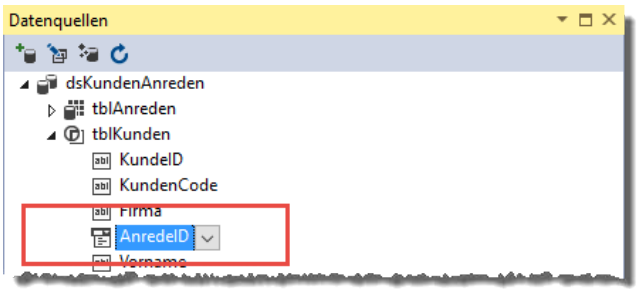


Bild 6: Das Feld **AnredeID** sollte nun als Kombinationsfeld im Fenster landen.

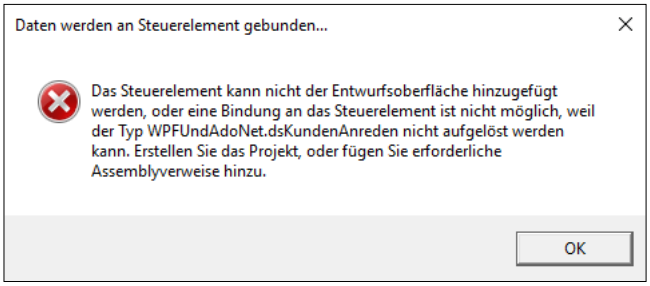


Bild 7: Eventuelle Fehlermeldung beim Versuch, Elemente aus dem Datenquellen-Dialog in ein WPF-Fenster zu ziehen

willkürlicher Reihenfolgt. Es scheint so, als ob dort nicht die Werte des Feldes **AnredeID** der Tabelle **tblAnreden**, sondern die der Tabelle **tblKunden** angezeigt werden (siehe Bild 9).

Kombinationsfelder mit Daten füllen

Unter Access ist das einfach: Fremdschlüsselfeld als Nachschlagefeld definieren, Tabelle an Formular binden, Fremdschlüsselfeld in den Entwurf ziehen – fertig ist das Kombinationsfeld zur Auswahl von Werten einer Lookup-Tabelle. Unter C#/WPF sieht das etwas anders aus. Dieser Grundlagenartikel zeigt, wie Sie ein Kombinationsfeld mit den Daten einer Lookup-Tabelle füllen und wie Sie auf wichtige Informationen wie etwa den Primärschlüsselwert des gewählten Datensatzes zugreifen – und das auf Basis von typisierten und untypisierten DataSets.

Unter Access/VBA konnten Sie selbst auf Basis einer reinen Tabelle schnell ein Kombinationsfeld erstellen, dass etwa das Feld **Anrede** aus **tblAnreden** anzeigte und das Primärschlüsselfeld **AnredeID** als gebundenes Feld nutzte. Unter WPF/C# sind dazu schon ein paar Zeilen Code nötig, damit es funktioniert.

Tabelle in ComboBox per C#

Die erste Variante, die wir uns dabei ansehen, basiert komplett auf C# (siehe Bild 1). Der XAML-Code, der das Aussehen des **ComboBox**-Steuerelements liefert, weiß nichts davon, dass das Steuerelement Daten aus einer Tabelle einer Datenbank anzeigen soll.

Dementsprechend kommt die Definition dieses **ComboBox**-Steuerelements sehr sparsam daher:

```
<ComboBox x:Name="cboAnreden" HorizontalAlignment="Left"
Margin="95,45,0,0" VerticalAlignment="Top" Width="145" />
```

Der C#-Code, der beim Laden des Fensters ausgeführt werden soll und der für das Füllen des **ComboBox**-Steuerelements verantwortlich ist, fällt dafür umso größer aus (siehe Listing 1). Damit diese Methode aufgerufen wird, fügen wir dem **Window**-Element die Eigenschaft **Loaded** mit dem Wert **Window_Loaded** hinzu (**Loaded** eingeben, zwei Mal auf die Tabulator-Taste drücken). Das **Window**-Element sieht dann etwa so aus:

Bild 1: Einfaches Kombinationsfeld mit den Daten der Tabelle **tblAnreden**

```
<Window Title="MainWindow" ... Height="350" Width="525" Wi
ndowStartupLocation="CenterScreen" Icon="data_table.ico"
Loaded="Window_Loaded">
    ...
</Window>
```

Die Methode **Window_Loaded** erledigt zunächst nichts Anderes, als ein untypisiertes DataSet zu erstellen – und zwar auf Basis der Tabelle **tblAnreden** der Access-Datenbank namens **Suedsturm.mdb**, die im gleichen Verzeichnis wie die **.exe**-Datei der erstellten Anwendung liegen soll.

> Im Beitrag **Datenzugriff mit ADO.NET, Teil 2** erfahren Sie, wie untypisierte **DataSet**-Objekte funktionieren und wie Sie **DataTable**-Objekte nutzen.

Wichtig ist, dass wir am Ende ein **DataTable**-Objekt mit den gewünschten Daten der Tabelle **tblAnreden** erhalten. Dieses liefert über die Eigenschaft **DefaultView** eine Form, in der wir es an die Eigenschaft **ItemsSource** des **ComboBox**-Steuerelements übergeben können. Dies entspricht grundsätzlich dem Zuweisen einer Tabelle an die Eigenschaft **RecordSource** unter Access/VBA. Dort haben Sie für ein Kombinationsfeld über die Eigenschaften **Gebundene Spalte**, **Spaltenanzahl** und **Spaltenbreiten** festgelegt, welche

```
private void Window_Loaded(object sender, RoutedEventArgs e) {
    DataTable dtAnreden = new DataTable();
    string strConnection = "Provider=Microsoft.Jet.OLEDB.4.0;Data Source=Suedsturm.mdb";
    string strSQL = "SELECT * FROM tblAnreden";
    OleDbConnection cnn = new OleDbConnection(strConnection);
    OleDbCommand cmd = new OleDbCommand();
    OleDbDataAdapter da;
    DataSet ds = new DataSet();
    DataTable dt;
    cmd.Connection = cnn;
    cmd.CommandText = strSQL;
    da = new OleDbDataAdapter(strSQL, cnn);
    da.Fill(ds, "tblAnreden");
    dt = ds.Tables["tblAnreden"];
    cboAnreden.ItemsSource = dt.DefaultView;
    cboAnreden.DisplayMemberPath = "Anrede";
    cboAnreden.SelectedValuePath = "AnredeID";
}
```

Listing 1: Füllen einer ComboBox per C#

Spalte als Wert des Kombinationsfeldes dienen und welche angezeigt werden sollte. Das ist unter C# etwas gezielter formuliert: Hier nutzen Sie die Eigenschaft **DisplayMemberPath**, um den Namen der anzuzeigenden Spalte zu übergeben, in diesem Fall das Feld **Anrede**.

Die Eigenschaft **SelectedValuePath** hingegen nimmt das Feld entgegen, das man unter Access mit dem Index der gebundenen Spalte festgelegt hätte – in diesem Fall **AnredeID**.

Angezeigten Wert und tatsächlichen Wert auslesen

Damit ist das Kombinationsfeld bereits ausgestattet. Schauen wir uns nun an, wie Sie per Code auf die beiden Werte für den aktuell markierten Eintrag des **ComboBox**-Steuerelements zugreifen können. Dazu fügen Sie der XAML-Definition des **ComboBox**-Steuerelements das Attribut **SelectionChanged** hinzu und legen gleich noch die entsprechende Methode an:

```
<ComboBox x:Name="cboAnreden" ...
SelectionChanged="cboAnreden_SelectionChanged"/>
```

Die Methode ergänzen Sie wie in Listing 2. Dort deklarieren wir zunächst eine Variable des Typs **ComboBox** und weisen dieser den mit dem Parameter **sender** gelieferten Verweis auf das auslösende Steuerelement zu.

Dann deklariert die Methode ein Objekt des Typs **DataRowView**, was einem Datensatz der **DataView** entspricht,

```
private void cboAnreden_SelectionChanged(object sender, System.Windows.Controls.SelectionChangedEventArgs e) {
    ComboBox cbo = sender as ComboBox;
    DataRowView vw = cbo.SelectedItem as DataRowView;
    string anrede = vw["Anrede"].ToString();
    MessageBox.Show("Angezeigter Wert: " + cboAnreden.SelectedValue.ToString() + "\nAngezeigter Text: " + anrede);
}
```

Listing 2: Ausgabe von Wert und angezeigtem Text für den ausgewählten Eintrag eines Kombinationsfeldes

die Sie dem **ComboBox**-Steuerelement als **ItemsSource** zugewiesen haben. Diese füllen Sie mit dem Objekt, das die Eigenschaft **SelectedItem** des Kombinationsfeldes liefert – zuvor mit **as DataRowView** explizit in den entsprechenden Datentyp konvertiert.

Den angezeigten Wert erhalten Sie dann über das Feld **Anrede** des **DataRowView**-Objekts, den tatsächlichen Wert des mit **SelectedValuePath** angegebenen Feldes liefert die Eigenschaft **SelectedValue** des Kombinationsfeldes. Beides gibt die Methode in einem Meldungsfenster aus (siehe Bild 2).

Bild 2: Ausgabe des Wertes und des angezeigten Textes eines Kombinationsfeldes per Meldungsfenster

ComboBox aus Eigenschaft füllen

Im zweiten Beispiel wollen wir so viele XAML-Attribute wie möglich verwenden, um das **ComboBox**-Steuerelement mit Daten zu füllen, und so die Funktionalität vom C#-Code nach WPF verlagern.

Dazu fügen Sie einem neuen Element dieses Typs namens **cboAnreden_WPF** hinzu und ergänzen einige Attribute:

```
<ComboBox x:Name="cboAnreden_DataView" ...
    DisplayMemberPath="Anrede"
    SelectedValuePath="AnredeID"
    ItemsSource="{Binding Path=dvAnreden}"/>
```

DisplayMemberPath und **DisplayValuePath** übernehmen wir einfach von der C#-Version des vorherigen Beispiels. **ItemsSource** benötigen wir auch, allerdings können wir

```
public DataView dvAnreden {
    get {
        DataTable dtAnreden = new DataTable();
        string strConnection = "Provider=Microsoft.Jet.OLEDB.4.0;Data Source=Suedsturm.mdb";
        string strSQL = "SELECT * FROM tblAnreden";
        OleDbConnection cnn = new OleDbConnection(strConnection);
        OleDbCommand cmd = new OleDbCommand();
        OleDbDataAdapter da;
        DataSet ds = new DataSet();
        DataTable dt;
        cmd.Connection = cnn;
        cmd.CommandText = strSQL;
        da = new OleDbDataAdapter(strSQL, cnn);
        da.Fill(ds, "tblAnreden");
        dt = ds.Tables["tblAnreden"];
        return dt.DefaultView;
    }
}
```

Listing 3: Eigenschaft, um ein **ComboBox**-Element mit den Daten eines **DataTable**-Objekts zu füllen

SQL Server 2014 Express installieren

Eine .NET-Anwendung kann man als Frontend für eine Access-Datenbank nutzen. Den vollen Nutzen erhält man allerdings erst, wenn man beispielsweise den SQL Server 2014 als Backend verwendet. Also schauen wir uns den SQL Server 2014, die derzeit aktuelle Version des relationalen Datenbankmanagement-Systems von Microsoft, etwas genauer an. In diesem Artikel erfahren Sie, welche Editionen es gibt und wie Sie den SQL Server 2014 auf Ihrem Rechner installieren.

SQL Server 2014-Editionen

Der SQL Server 2014 kommt in den folgenden Editionen:

- SQL Server 2014 Express: Diese Version ist kostenlos und kann beliebig eingesetzt werden. Sie hat allerdings ein paar Einschränkungen. Auf der technischen Seite sind dies beispielsweise die Begrenzung auf den nutzbaren Arbeitsspeicher von einem Gigabyte, vier Prozessorkerne und eine maximale Datenbankgröße von zehn Gigabyte. Gegenüber den zwei Gigabyte einer Access-Datenbank ist dies schon eine beachtliche Steigerung.
- SQL Server 2014 Enterprise, Business Intelligence und Standard: Enthalten je nach Version weitere Features, die wir aber im Rahmen dieses Magazins vorerst nicht benötigen.

Weitere Informationen zu den verschiedenen Editionen finden Sie unter **https://www.microsoft.com/de-de/server-cloud/products/sql-server-editions/overview.aspx**.

Damit ist schnell klar: Wenn Sie den SQL Server 2014 benötigen, um mit Datenbankanwendungen auf Basis von .NET zu experimentieren, reicht die kostenlose SQL Server 2014 Express-Edition völlig aus. Deren Installation beschreiben wir in den folgenden Abschnitten.

SQL Server 2014 Express-Versionen

Den Download finden Sie, wenn Sie von oben genanntem Link aus unter **SQL Server 2014 Express** auf **Weitere Informationen** klicken. Hier folgt eine weitere Herausforderung in Form verschiedener Versionen:

- **SQL Server Express mit Tools**: Enthält nur die Verwaltungstools.
- **SQL Server Management Studio**: Enthält nur die Management-Tools für den SQL Server, nicht aber das Datenbankmanagement-System selbst.
- **SQL Server LocalDB (MSI-Installationsprogramm)**: Schnell zu installierende Version des SQL Servers in einer einfachen Variante.
- **SQL Server Express mit Advanced Services**: Paket mit SQL Server 2014, Reporting Services, SQL Server Management Studio und mehr.
- **SQL Server Express**: SQL Server ohne Tools.

Die Developer Edition

Neben den oben genannten Editionen gibt es auch noch die Developer Edition. Dabei handelt es sich um eine voll ausgestattete SQL Server-Version, die Sie zu einem recht günstigen Preis erwerben können (maximal um die 100 EUR). Diese darf jedoch nur zum Entwickeln von Anwendungen genutzt werden. Sollten Sie also damit experimentieren wollen und Sie benötigen Funktionen, welche die Express Edition nicht liefert, können Sie auf die Developer Edition zurückgreifen.

Download von SQL Server 2014 Express

Wir gehen davon aus, dass Sie den bequemen Weg über den Download der gewünschten Version des SQL Servers wählen. Alternativ haben Sie diese möglicherweise schon als DVD vorliegen und installieren den SQL Server von dort aus.

Bild 1: Neu oder Update?

Bild 2: Akzeptieren der Lizenzbedingungen und Übersicht

Bevor Sie mit dem Download beginnen, müssen Sie sich mit Ihrem Microsoft-Konto einloggen oder ein neues Konto anlegen. Danach wählen Sie die gewünschte Version aus.

Wir gehen auf Nummer sicher und verwenden die Version **SQL Server 2014 Express with Advanced Services 64bit** (für Rechner mit 64bit-Betriebssystem). Dies lädt, sofern Sie die deutsche Version gewählt haben, die Datei **SQLEXPRADV_x64_DEU.exe** herunter – mit immerhin rund 1.200 Megabyte Größe.

Das Setup fragt zunächst, ob Sie eine neue Installation vornehmen möchten oder ob Sie ein Update einer bereits auf dem Rechner vorhandenen Version durchführen möchten (siehe Bild 1).

Im zweiten Schritt akzeptieren Sie die Lizenzbedingungen und entscheiden sich nach Wunsch dafür, Microsoft durch entsprechendes Feedback bei der Weiterentwicklung des SQL Servers zu unterstützen (siehe Bild 2).

Der Schritt **Microsoft Update** prüft, ob Aktualisierungen für die aktuelle Windows-Installation und auch für die zu installierende SQL Server-Software vorliegen und fragt, ob Sie diese herunterladen wollen.

Installationsoptionen

Direkt im Anschluss folgt der wichtige Schritt aus Bild 3. Hier legen Sie unter anderem fest, mit welchen Funktionen SQL Server installiert werden soll. Wenn Sie genügend Speicherplatz haben, sollten Sie direkt alle Optionen, die standardmäßig angeboten werden, beibehalten. Auf die **SQL Server Replikation** und die **Volltext- und semantische Extraktion für die Suche** können Sie gegebenenfalls verzichten, zumindest werden wir in diesem Magazin wohl nicht auf diese Funktionen eingehen. Die **Reporting Services** werden wir früher oder später sicher ansprechen, da – wenn Sie von Access zu .NET wechseln – ja auch die von Access gewohnten Berichte

wegfallen. Von den übrigen Funktionen ist vor allem der Eintrag **Verwaltungstools – Vollständig** interessant, da Sie damit nicht nur das Management Studio erhalten, sondern auch den Profiler und Unterstützung für die Reporting Services.

Klicken Sie auf die einzelnen Einträge unter **Funktionen**, damit unter **Funktionsbeschreibung** Details zu der jeweiligen Funktion eingeblendet werden.

Schließlich legen Sie unter **Instanzstammverzeichnis** noch fest, in welchem Bereich des Dateisystems die Installation erfolgen soll.

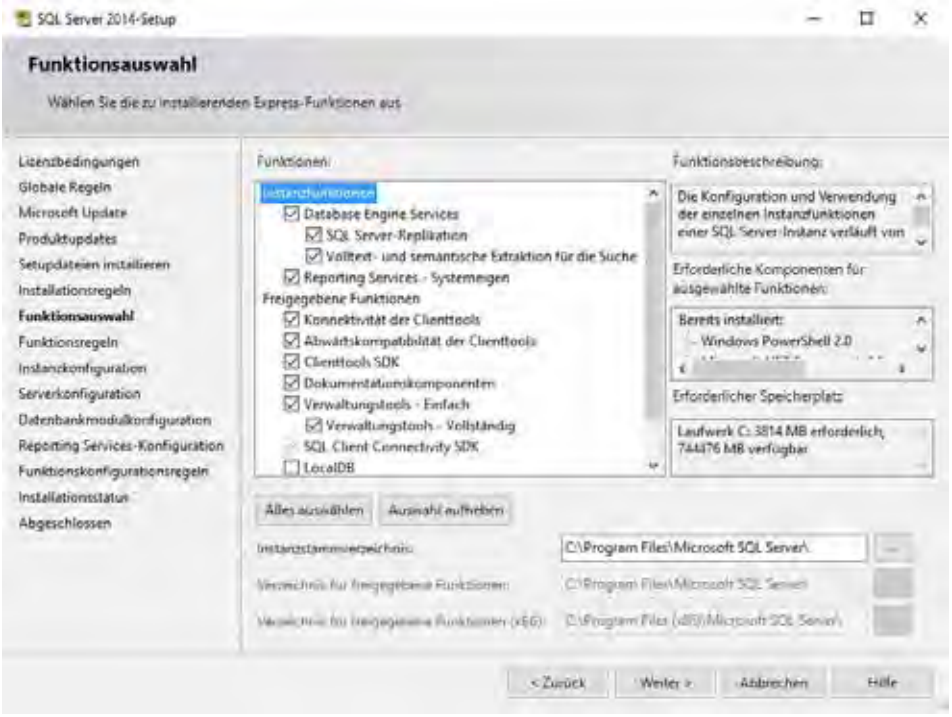


Bild 3: Festlegen der Installationsoptionen

Standardinstanz oder benannte Instanz

Nach ein paar Sekunden Verschnaufpause meldet sich der Installationsassistent wieder zurück und stellt den Dialog aus Bild 4 bereit.

Hier legen Sie nun fest, ob Sie den SQL Server als Standardinstanz oder als benannte Instanz installieren möchten. Im Gegensatz zu früheren Varianten stellt der Installationsassistent den Standardwert auf **Benannte Instanz** ein und schlägt als Namen **SQLEXPRESS** vor. Dies ist sinnvoll, wenn Sie mehrere Instanzen des SQL Servers auf einem Rechner laufen lassen möchten. Wenn Sie also etwa schon einen SQL Server betreiben und für Experimente mit

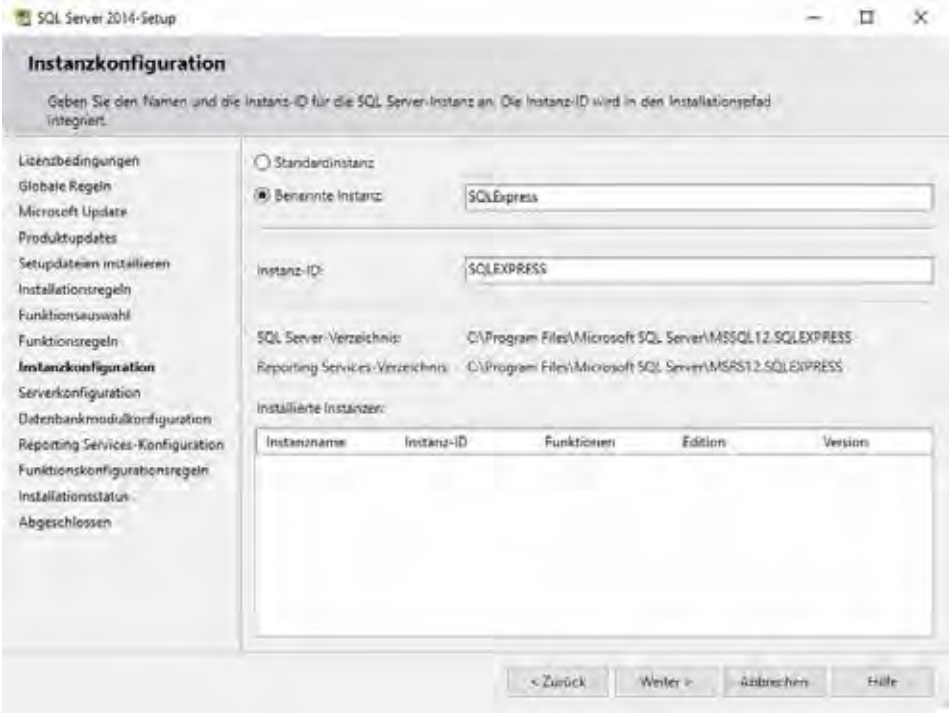


Bild 4: Installieren einer Standardinstanz oder einer benannten Instanz

Von der .mdb-Datei zum SQL Server

Der Datenzugriff von .NET auf eine .mdb-Datei ist möglich, aber die vollen Möglichkeiten erlaubt erst beispielsweise eine Datenbank, die mit dem SQL Server verwaltet wird. Da viele Leser wohl von der Access-Schiene kommen, dürfte der eine oder andere seine Datenbank zum SQL Server transferieren wollen, um damit von einer .NET-Anwendung aus zu experimentieren. Dieser Artikel zeigt, wie Sie eine Migration der Tabellen und Abfragen einer Anwendung schnell realisieren.

Voraussetzung für die Durchführung der hier vorgestellten Techniken ist das Vorhandensein einer zu migrierenden Access-Datenbank sowie einer Instanz des SQL Servers.

Außerdem wollen wir den **SQL Server Migration Assistent for Access** (kurz **SSMA**) nutzen, um die Migration durchzuführen. Diesen laden Sie von der folgenden Webseite herunter:

https://www.microsoft.com/en-us/download/confirmation.aspx?id=43690

Nach dem Download starten Sie direkt den Setup-Assistenten des SSMA. Die dazu nötigen Schritte sind selbsterklärend.

Start des SQL Server Migration Assistants

Der Assistent wird standardmäßig mit dem SSMA aufgerufen und liefert vorab eine Liste der anstehenden Aufgaben (siehe Bild 1).

Im ersten Schritt fragt der Assistent dann direkt den Namen des zu erstellenden SSMA-Projekts ab (nicht zu verwechseln mit dem Namen der zu erstellenden Datenbank) sowie den Speicherort der Projektdatei (siehe Bild 2).

Bild 1: Startdialog des SSMA-Assistenten

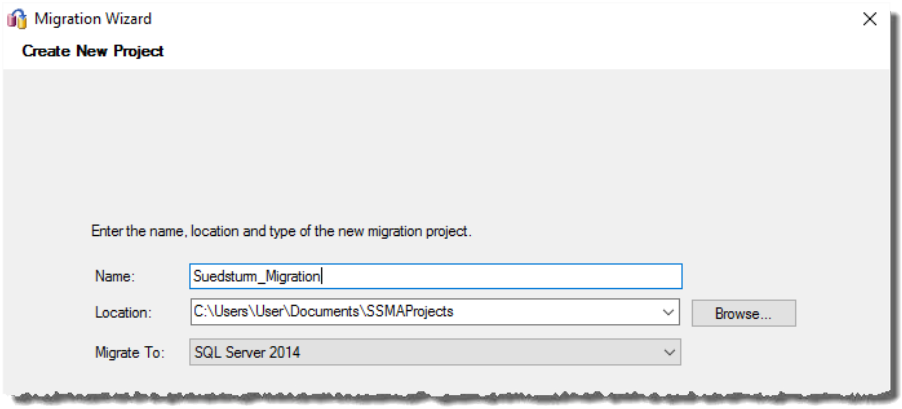


Bild 2: Angabe des Namens des Migrationsprojekts und des Speicherortes

Quelldatenbank auswählen

Der folgende Schritt des Assistenten fragt dann den Namen der Access-Datenbank ab, deren Tabellen in eine SQL Server-Datenbank übertragen werden sollen (siehe Bild 3). Hier wählen wir die Datei **Suedsturm.mdb** aus, da wir deren Tabellen für folgende Artikel als Beispieldatenbank verwenden wollen.

Danach folgt die Auswahl der zu migrierenden Tabellen und Abfragen. Wir wollen an dieser Stelle nur Tabellen übertragen, also aktivieren wir nur den Haken des Eintrags **Tabellen** und erweitern die darunter

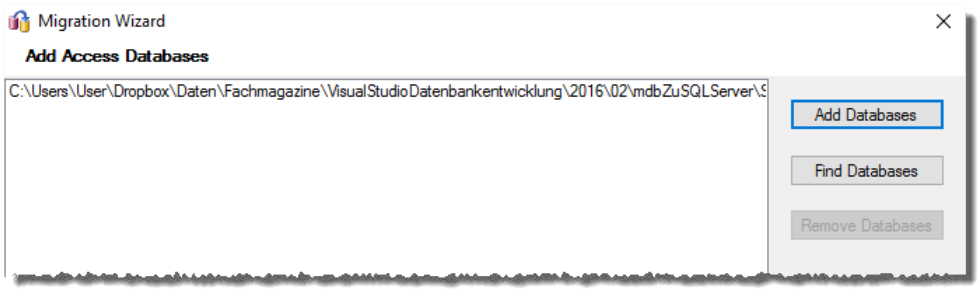


Bild 3: Hinzufügen der zu migrierenden Access-Datenbank

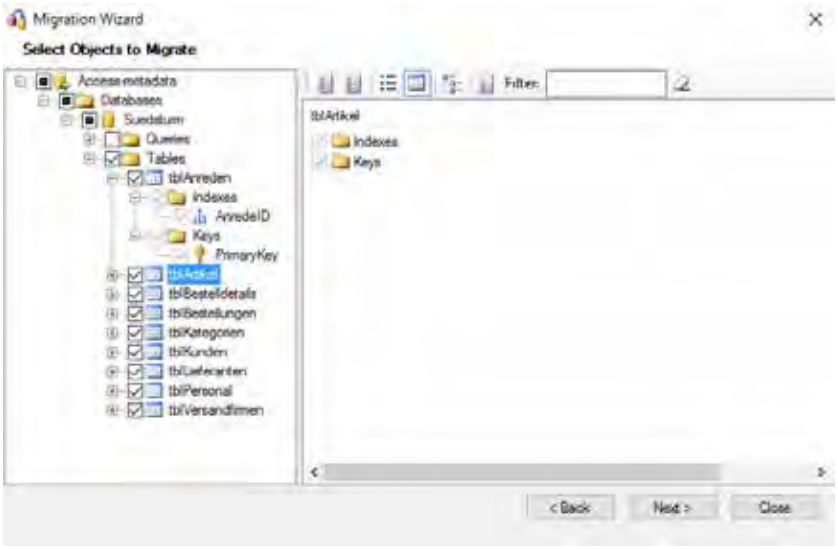


Bild 4: Auswahl der zu migrierenden Tabellen und Felder sowie ihrer Eigenschaften

befindlichen Elemente (die Migration von Abfragen ist auch nicht sonderlich ergiebig). Im rechten Bereich sehen Sie dann die Eigenschaften der Tabellen, insbesondere der Indizes und Schlüsselfelder.

Mögliche Probleme

Bei meiner Konstellation (Windows 10, Office 2016 32bit) gab es ein Problem beim Zugriff auf die zu migrierende Access-Datenbank, das sich im Auftauchen eines Ausrufezeichens wie in Bild 5 zeigte. Dieses tritt manchmal auf, wenn man die falsche Version des SSMA verwendet, also die 32bit-Variante mit dem 64bit-Office oder die 64bit-Variante mit dem 32bit-Office. Sollten bei Ihnen im Migrationsassistenten keine Tabellen angezeigt werden und ein Ausrufezeichen im Icon der Datenbank erscheinen, ist dies möglicherweise der Fall. Achten Sie dann darauf, die Version des SSMA zu starten, die zu Ihrem Office-Paket passt (also etwa SSMA 32bit zu Office 32bit).

Das Problem kann noch eine weitere Ursache haben. Dabei fehlen bestimmte Komponenten beziehungsweise sie liegen nicht in der richtigen Version vor. Laden Sie dann die Komponente **Microsoft Access Database Engine 2010 Redistributable** von folgender Internetadresse herunter:

https://www.microsoft.com/de-DE/download/details.aspx?id=13255

Anschließend sollte der Fehler nicht mehr auftreten. Die Komponente ist zwar für 2010 ausgezeichnet, aber es funktioniert bei mir auch unter Access 2016.

Ziel festlegen

Im folgenden Schritt geben Sie den Namen der SQL Server-Instanz ein, in der die neue Datenbank erstellt werden soll. Leider zeigt der Dialog aus Bild 6 in unserem Fall keine Liste der verfügbaren SQL Server an, sodass wir diesen manuell eintragen mussten. Auch erschien nach der Eingabe keine Liste der vorhandenen Datenbank, was aber keine Rolle spielte, da wir diese ja ohnehin neu erstellen wollten. Wir haben der Datenbank den Namen **Suedsturm_SQL** gegeben.

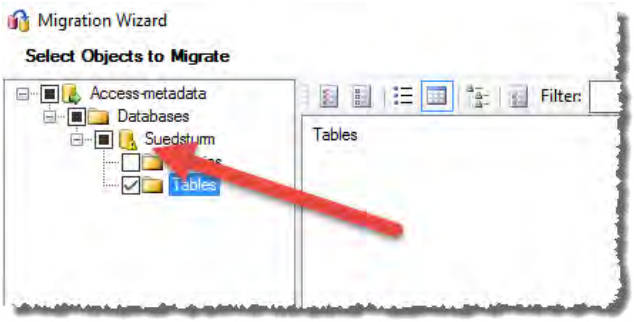


Bild 5: Problem mit dem Einlesen der Datenbank

Da die Datenbank zu diesem Zeitpunkt noch nicht angelegt wurde, fragt der SSMA-Assistent noch, ob diese neu erstellt werden soll (siehe Bild 7).

Bild 6: Festlegen der SQL Server-Instanz und der Zieldatenbank

Verknüpfungen erstellen?

Auf Wunsch können Sie nun festlegen, ob in der Access-Datenbank Verknüpfungen zu den in der SQL Server zu erstellenden Tabellen hinzufügt werden sollen. Eigentlich ist das nicht nötig, da wir ja von .NET-Anwendungen auf die Tabellen zugreifen wollen.

Allerdings kann es auch nicht schaden, gelegentlich über ein Access-Frontend auf die Tabellen zuzugreifen – etwa, um zu prüfen, ob eine Änderung an den Daten wie gewünscht durchgeführt wurde. In diesem Fall aktivieren Sie die Option **Link Tables** des Dialogs aus Bild 8.

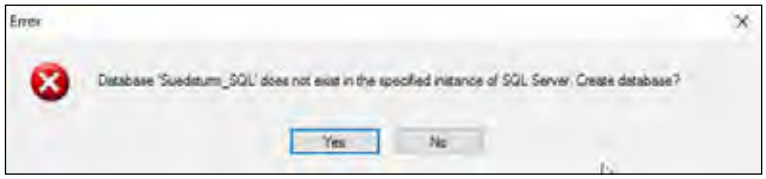


Bild 7: Soll die Datenbank neu erstellt werden?

Migration durchführen

Danach beginnt die eigentliche Migration. Das Ergebnis zeigt der Dialog **Migration Status** (siehe Bild 9). Im Falle der Migration der Beispieldatenbank traten keinerlei Fehler und lediglich einige Warnungen und Hinweise auf. Die vier dargestellten Bereiche haben dabei folgende Bedeutung:

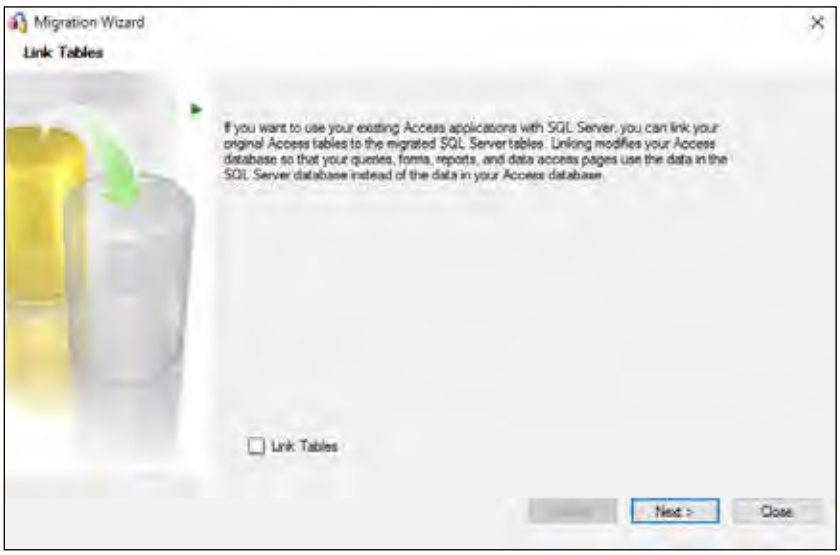


Bild 8: Sollen die Tabellen verknüpft werden?

- **Convert selected objects**: Umwandlung der Objekte in Form eines Skripts zur Definition von Tabellen- und Abfragestruktur
- **Load converted objects into the target database**: Tabellen und Sichten in der SQL Server-Datenbank erstellen
- **Migrate data for selected objects**: Migration der in den Tabellen enthaltenen Daten
- **Link converted tables**: Erstellen der Verknüpfungen zu den neu erstellten Tabellen in der Access-Datenbank

ADO.NET: Typisierte und untypisierte DataSets

Unter ADO.NET gibt es typisierte und untypisierte DataSets. In den bisherigen Artikeln haben Sie meist mit untypisierten DataSets gearbeitet. Das heißt, dass Sie alle benötigten Objekte per Code erzeugt und dabei auch Verbindungseigenschaften, SQL-Anweisungen et cetera übergeben haben. Bei typisierten DataSets definieren Sie die meisten Eigenschaften des später im Code verwendeten DataSets über einen Assistenten. Dies gelingt durch einige Klassen, die vom Assistenten automatisch erzeugt werden und mit deren Hilfe Sie später direkt per IntelliSense auf die enthaltenen Tabellen und Felder zugreifen können.

Doch eins nach dem anderen: Wir wollen zunächst einmal den zuständigen Assistenten bemühen, um ein typisiertes DataSet auf Basis zweier Tabellen der Beispieldatenbank **Suedsturm.mdb** zu erzeugen. Später zeigen wir dann, welche Möglichkeiten das typisierte DataSet gegenüber seinem nicht typisierten Pendant liefert.

Datenquelle hinzufügen

Wir wollen die beiden Tabellen **tblArtikel**, **tblKategorien** und **tblLieferanten** zum DataSet hinzufügen. Um dies zu erledigen, zeigen Sie zunächst mit dem Menüeintrag **Ansicht|Server**-Explorer den Server-Explorer an, der auch einen Eintrag namens **Datenverbindungen** enthält. Wählen Sie den Eintrag **Verbindung hinzufügen...** aus dem Kontextmenü dieses Eintrags aus.

Es erscheint nun der Dialog Datenquelle auswählen, der verschiedene Datenquellentypen anzeigt. Wählen Sie hier den Typ **Microsoft Access-Datenbankdateien** aus und klicken Sie auf **Weiter** (siehe Bild 1).

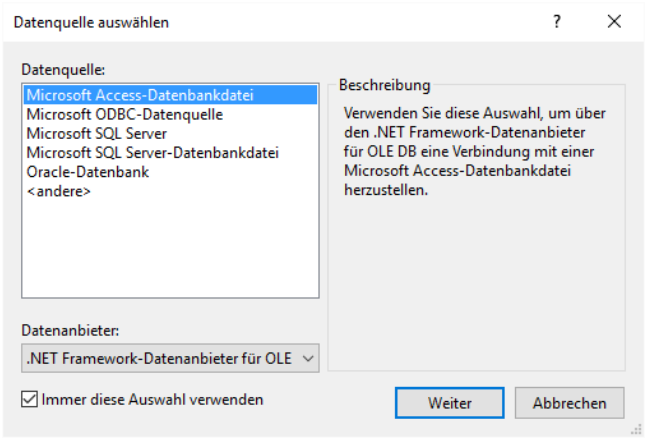


Bild 1: Auswählen des Typs der Datenquelle

Im folgenden Schritt geben Sie die konkrete Access-Datenbankdatei an. Testen Sie die Verbindung, indem Sie auf Testverbindung klicken. Liefert dies keinen Fehler, beenden Sie den Assistenten mit einem Klick auf die Schaltfläche **OK** (siehe Bild 2).

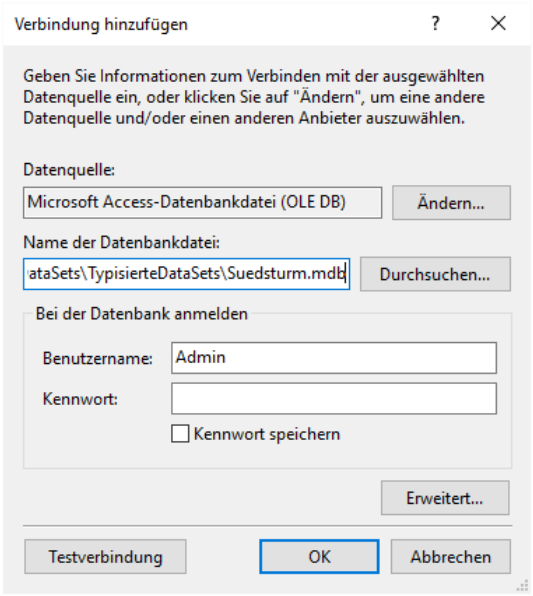


Bild 2: Auswählen der Datenquelle

Im Anschluss finden Sie im Server-Explorer unter Datenverbindungen einen neuen Eintrag namens **Suedsturm.mdb** vor. Wenn Sie diesen Eintrag erweitern, finden Sie schnell heraus, dass auch alle enthaltenen Tabellen zur Verfügung stehen (siehe Bild 3).

Typisiertes DataSet Erstellen

Wählen Sie nun den Menüeintrag **Projekt|Neues Element hinzufügen...** aus, erscheint der Dialog

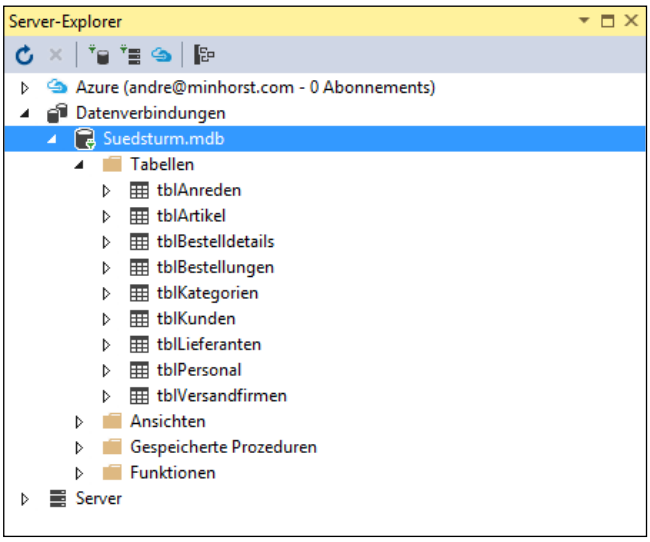


Bild 3: Die hinzugefügte Datenquelle im Server-Explorer

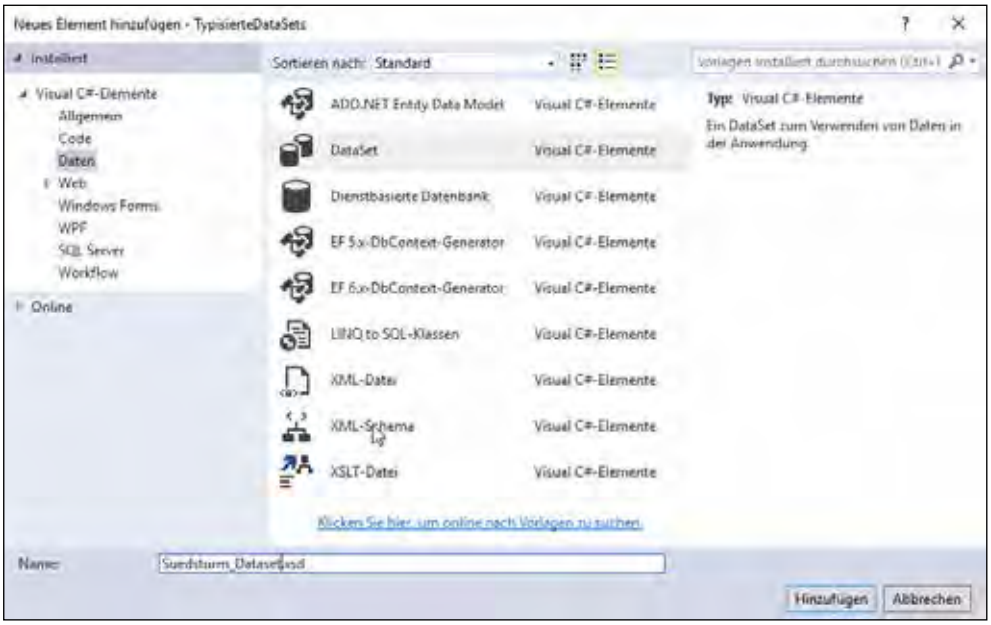


Bild 4: Hinzufügen eines **DataSet**-Objekts zum Projekt

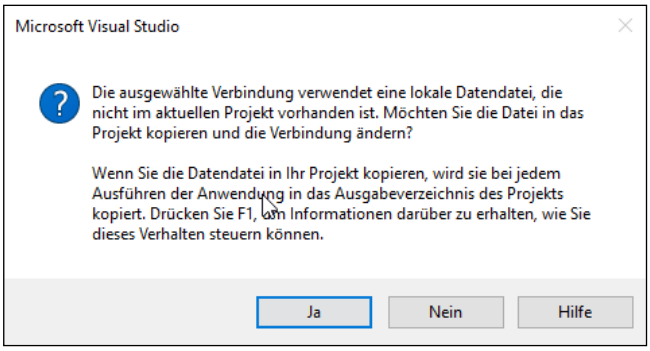


Bild 5: Meldung beim Hinzufügen der ersten Tabelle der Datenquelle

Neues Element hinzufügen. Mit diesem wählen Sie nun die Kategorie **Visual C#-ElementelDaten** aus und klicken dann auf das Element **DataSet**. Tragen Sie als Name **Suedsturm_DataSet.xsd** ein und klicken Sie auf **Hinzufügen** (siehe Bild 4).

Das neue Objekt **Suedsturm_DataSet.xsd** wird nun im Designer in Visual Studio angezeigt. Aktivieren Sie nun den Server-Explorer, sodass dieser neben dem neuen Objekt **Suedsturm_DataSet.xsd** zu sehen ist.

Nun ziehen Sie die Tabellen **tblArtikel**, **tblLieferanten** und **tblKategorien** aus dem Server-Explorer in den Entwurf des DataSets. Direkt beim Versuch, die erste Tabelle hinüberzuziehen, sollte die Meldung aus Bild 5 erscheinen. Wenn Sie hier auf Ja klicken, wird die verwendete Datenbankdatei **Suedsturm.mdb** zum Projekt hinzugefügt.

Danach geht es los: Ziehen Sie die drei Tabellen in den Entwurf des DataSets und Sie erhalten die Konstellation aus Bild 6. Die Tabellen werden mit allen enthaltenen Feldern genau wie im Beziehungs-Fenster von Access angezeigt – nicht einmal die Beziehungspfeile fehlen. Ein wichtiger Unterschied ist jedoch, dass jede Tabelle unten ein weiteres Objekt etwa namens **tblArtikelTableAdapter** enthält, dass die beiden Methoden **Fill** und **GetData** anbietet.

Nicht alle Felder einer Tabelle verwenden

Wie Sie sehen, wurden die Tabellen vollständig in das DataSet übernommen – wie soll es auch anders sein, wenn Sie die kompletten Tabellen hinüber ziehen? Andererseits kann es ja auch einmal vorkommen, dass Sie nicht alle Felder

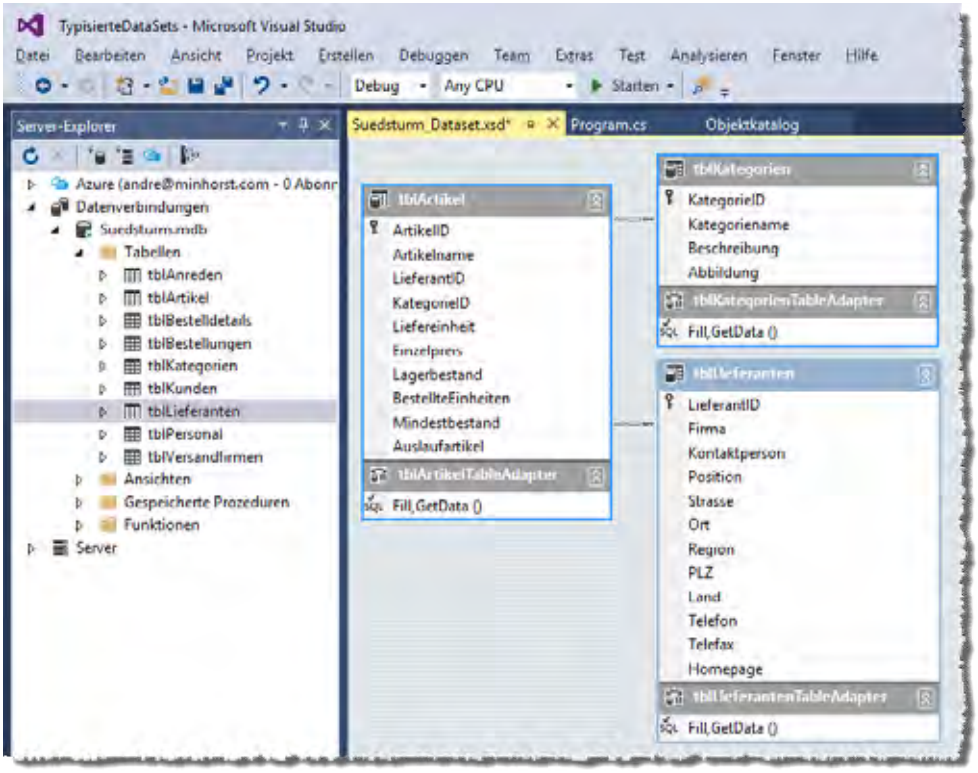


Bild 6: Das DataSet mit allen drei benötigten Tabellen

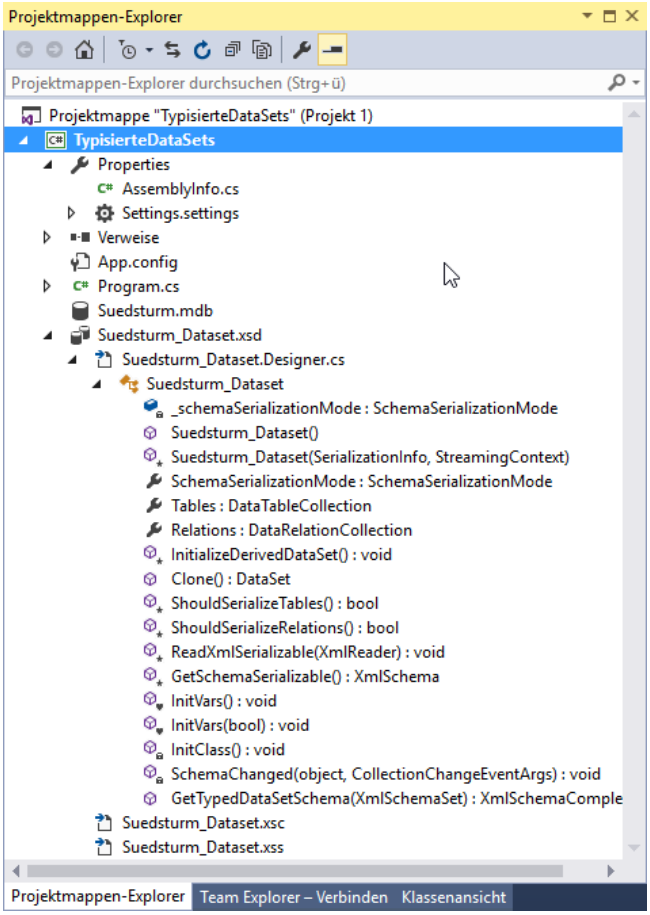


Bild 7: Neue Elemente im Projektmappen-Explorer

einer Tabelle verwenden möchten. In diesem Fall klappen Sie das Tabellen-Element im Server-Explorer so auf, dass Sie die einzelnen Feldnamen der Tabelle sehen. Hier wählen Sie nun entweder bei gedrückter Umschalt- oder **Strg**-Taste die gewünschten Felder aus. Bei gedrückter Umschalttaste markieren Sie alle Felder zwischen dem zuerst und dem zuletzt angeklickten Feld, mit gedrückter **Strg**-Taste aktivieren oder deaktivieren Sie einzelne Einträge. In diesem Fall wollen wir jedoch alle Felder der Tabellen nutzen.

Vom Assistenten erzeugte Elemente

Damit ist das typisierte DataSet bereits fertiggestellt – nun schauen wir uns an, was der Assistent im Hintergrund alles für uns erledigt hat. Dazu brauchen Sie den Blick nur zum Projektmappen-Explorer zu wenden (siehe Bild 7). Hier finden Sie nun einen neuen Eintrag namens **Suedsturm_DataSet.xsd** mit dem Unterelement **Suedsturm_DataSet.Designer.cs**. Mit einem Doppelklick auf ersteren gelangen Sie später zum Designer zurück, um beispielsweise Tabellen oder Felder hinzuzufügen oder zu entfernen.

Der Eintrag **Suedsturm_DataSet.Designer.cs** zeigt auf die neu erstellte Klasse, welche im aktuellen Zustand zunächst die ein paar der Elemente bereithält, die auch ein untypisiertes **DataSet**-Objekt liefert – beispielsweise die **Tables**- oder die **Relations**-Auflistung oder auch einige Methoden. Allerdings ist hier noch nichts zu sehen von den für unsere Zusammenstellung der drei Tabellen **tblArtikel**, **tblKategorien** und **tblLieferanten** typischen Elemente. Es gibt weder einen DataAdapter noch DataTables. Das ändert sich allerdings

schlagartig, sobald Sie das typisierte DataSet beispielsweise mit der Tastenkombination **Strg + S** speichern. Dann wird Visual Studio nämlich aktiv und erweitert die Klasse um eine ganze Reihe von Elementen. Die wichtigsten zeigt Bild 8:

- **Suedsturm_Dataset** ist eine Klasse, welche von der Klasse **DataSet** abgeleitet ist. Das heißt, es enthält alle Methoden, Eigenschaften und Ereignisse dieser Klasse, fügt aber eigene Elemente hinzu oder überschreibt Elemente der Klasse. Durch das Hinzufügen oder Überschreiben von Elementen, die sich speziell auf unsere Tabellen beziehen, erhalten wir ja eben das typisierte **DataSet**-Objekt.

- Die Klassen **tblArtikelTableAdapter**, **tblKategorienTableAdapter** und **tblLieferantenTableAdapter** sind von der Klasse **TableAdapter** abgeleitet.

Die Klasse **Suedsturm_DataSet** enthält noch die drei Klassen **tblArtikelDataTable**, **tblKategorienDataTable** und **tblLieferantenDataTable**. Für den Zugriff auf die einzelnen Zeilen der Tabellen stellen die drei Klassen **tblArtikelRow**, **tblKategorienRow** und **tblLieferantenRow** für jedes Feld der ursprünglichen Tabelle jeweils eine Eigenschaft zur

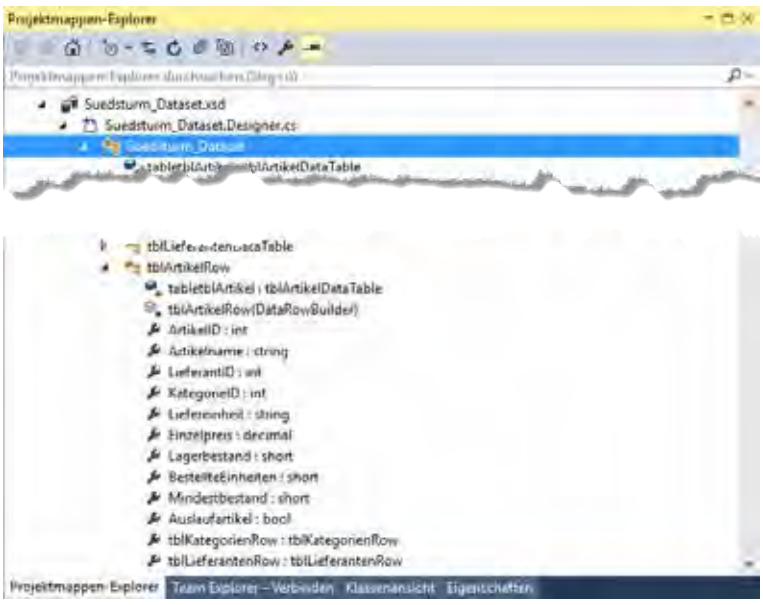


Bild 9: Für jedes einzelne Feld der beteiligten Tabellen gibt es eine in der jeweiligen Klasse definierte Eigenschaft.

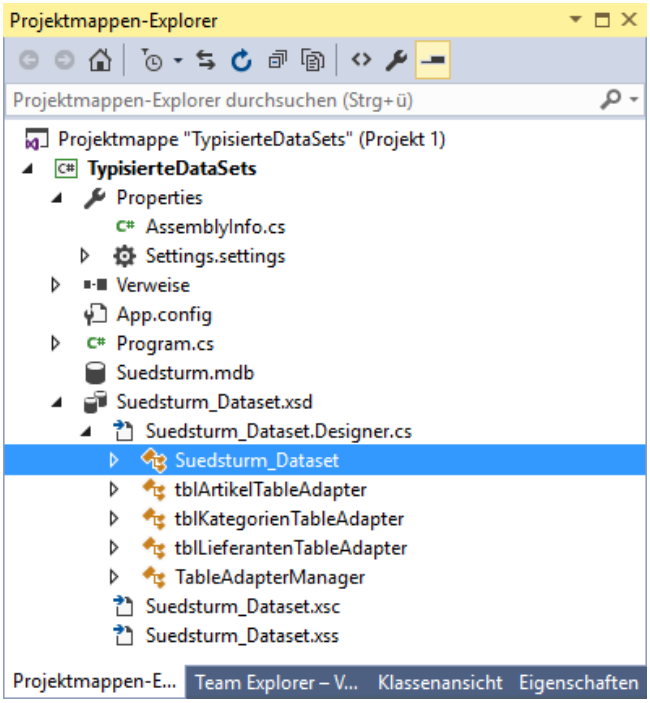


Bild 8: Die Hauptelemente des typisierten DataSets

Verfügung, über die Sie per Code auf die Feldinhalte des aktuellen Datensatzes zugreifen können (siehe Bild 9).

Beispiel der Ausgabe der Daten eines untypisierten DataSets

Schauen wir uns nun an, wie der Zugriff auf ein typisiertes DataSet im Gegensatz zu dem auf ein untypisiertes DataSet aussieht. Bei einem untypisierten DataSet würden Sie den Code aus Listing 1 verwenden. Hier stehen eine Reihe von Aufgaben an:

- Definieren des Connection-Strings

- Festlegen der SQL-Abfrage

- Deklarieren und gegebenenfalls Initialisieren von **Connection**-, **Command**-, **DataAdapter**-, **DataSet**- und **DataTable**-Objekten

- Zuweisen der **Connection**- und der **CommandText**-Eigenschaften des **Command**-Objekts

```
public static void DataSetUntypisiert() {
    string strConnection = "Provider=Microsoft.Jet.OLEDB.4.0;Data Source=Suedsturm.mdb";
    string strSQL = "SELECT * FROM tblArtikel";
    OleDbConnection cnn = new OleDbConnection(strConnection);
    OleDbCommand cmd = new OleDbCommand();
    OleDbDataAdapter da;
    DataSet ds = new DataSet();
    DataTable dt;
    cmd.Connection = cnn;
    cmd.CommandText = strSQL;
    da = new OleDbDataAdapter(strSQL, cnn);
    da.Fill(ds, "tblArtikel");
    dt = ds.Tables["tblArtikel"];
    foreach (DataRow dr in dt.Rows) {
        Console.WriteLine("{0} {1}", dr["ArtikelID"], dr["Artikelname"]);
    }
    Console.ReadLine();
}
```

Listing 1: Ausgabe der Daten eines untypisierten DataSets

```
public static void DataSetTypisiert() {
    Suedsturm_Dataset ds = new Suedsturm_Dataset();
    tblArtikelTableAdapter ta = new tblArtikelTableAdapter();
    ta.Fill(ds.tblArtikel);
    foreach (Suedsturm_Dataset.tblArtikelRow row in ds.tblArtikel) {
        Console.WriteLine("{0} {1}", row.ArtikelID, row.Artikelname);
    }
    Console.ReadLine();
}
```

Listing 2: Ausgabe der Daten eines typisierten DataSets

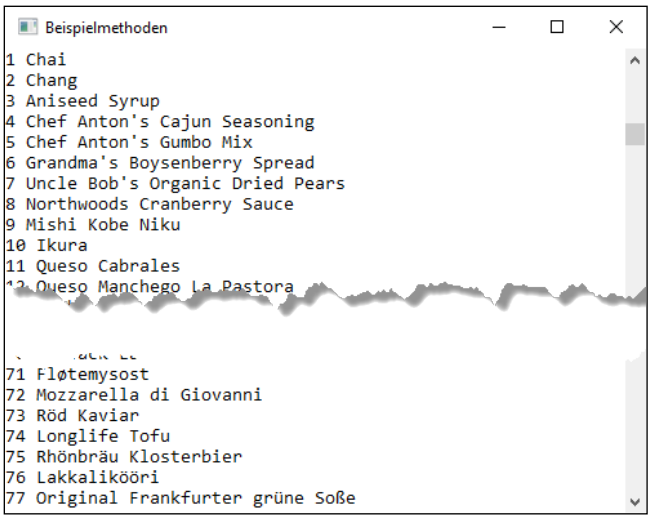


Bild 10: Ausgabe der Datensätze der Tabelle **tblArtikel** in der Console

- Erstellen und Füllen des **DataAdapter**-Objekts mit der Tabelle **tblArtikel**
- Zuweisen der Tabelle **tblArtikel** an die untypisierte **DataSet**-Variable
- Durchlaufen der **DataRow**-Objekte und Ausgabe der Werte in der Konsole wie in Bild 10 dargestellt

Beispiel der Ausgabe des typisierten DataSets

Listing 2 zeigt, wie Sie die gleiche Ausgabe der Daten der Tabelle **tblArtikel** in der Konsole mit dem typisierten DataSet erreichen. Hier haben wir mit dem Assistenten schon einige

Tipps und Tricks zu Fenstern und Steuerelementen

In dieser Reihe unserer Tipps und Tricks erfahren Sie Einiges über Fenster und Steuerelemente: Wie setzen Sie den Fokus auf ein Steuerelement? Wie legen Sie die Aktivierreihenfolge von Steuerelementen fest? Wie zentrieren Sie ein Fenster? Wie öffnen Sie ein Fenster relativ zum aufrufenden Fenster? Wie übergeben Sie Parameter beim Öffnen an ein Fenster? Wie legen Sie ein Anwendungssymbol fest, und wie eines für ein einzelnes Fenster? Mehr dazu auf den folgenden Seiten!

Fokus auf ein Steuerelement setzen

Im Gegensatz zu Access-Formularen landet der Fokus unter WPF nicht unbedingt sichtbar auf einem Steuerelement.

Daher schauen wir uns nun an, wie Sie den Fokus etwa beim Öffnen des Fensters auf das gewünschte Steuerelement verschieben. Dazu gibt es mindestens zwei Methoden: eine per XAML und eine per C#.

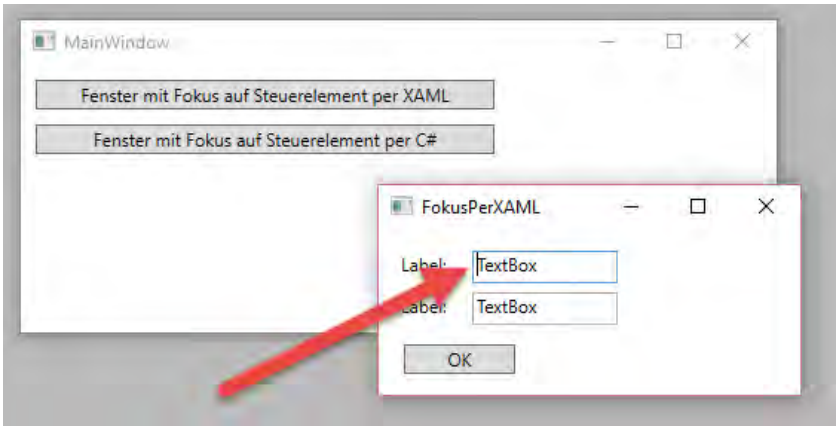


Bild 1: Öffnen eines Fensters und Setzen des Fokus

Um beide auszuprobieren, fügen wir dem Fenster **MainWindow** zwei Schaltflächen hinzu, mit denen wir die beiden weiteren Fenster **FokusPerXAML** und **FokusPerCSharp** öffnen. Beide sollen den Fokus jeweils auf das erste Textfeld verschieben (siehe Bild 1).

Unter XAML legen Sie in einem übergeordneten Element des zu betroffenen Elements fest, welches Element den Fokus erhalten soll – in diesem Fall im **Grid**-Element. Dort fügen Sie das Attribut **FocusManager.FocusedElement** hinzu und tragen als Wert **"{Binding ElementName=textBox}"** ein. Alles lässt sich einfach per IntelliSense hinzufügen:

```
<Window x:Class="TippsControls.FokusPerXAML" ...
                                    Title="FokusPerXAML">
    <Grid FocusManager.FocusedElement=
                          "{Binding ElementName=textBox}">
        <Label x:Name="label" Content="Label:" .../>
        <TextBox x:Name="textBox" Text="TextBox" .../>
        ...
    </Grid>
</Window>
```

Das Textfeld **textBox** erhält dann gleich nach dem Öffnen des Fensters den Fokus.

Unter C# ist es noch etwas einfacher. Hier definieren Sie mit dem Attribut **Loaded** den Ereignishandler, der beim Laden des Fensters aufgerufen werden soll:

```
<Window ... Loaded="Window_Loaded">
    ...
</Window>
```

Dann füge Sie der entsprechenden Methode eine einzige Anweisung hinzu, welche die Methode **Focus** des Steuerelements aufruft:

```
private void Window_Loaded(...) {
    this.textBox.Focus();
```

```
}
```

Damit verschiebt auch diese Variante beim Öffnen den Fokus auf das erste Textfeld.

Aktivierreihenfolge festlegen

Unter Access haben Sie die Aktivierreihenfolge in einem speziell dafür vorgesehenen Dialog festgelegt. Unter WPF läuft das etwas anders. Wenn Sie nichts weiter tun, entspricht die Aktivierungsreihenfolge der Reihenfolge, in der Sie die Elemente im XAML-Code angelegt haben. Solange diese Reihenfolge der gewünschten Aktivierreihenfolge entspricht, ist alles okay. Wenn Sie aber nun beispielsweise ein neues Textfeld plus Bezeichnungsfeld (**Label**) aus der Toolbox zwischen zwei vorhandene Textfelder einfügen, funktioniert die Aktivierreihenfolge nicht mehr wie gewünscht. Nun wird erst das erste, dann das dritte und schließlich das zweite Textfeld aktiviert. Um dies zu ändern, müssten Sie also die Reihenfolge der Elemente im XAML-Code anpassen.

Davon abgesehen, dass ein wenig Ordnung im XAML-Code nicht schaden kann, kann es natürlich Gründe geben, eine Aktivierreihenfolge vorzugeben, die nicht von der Gestaltung des XAML-Codes abhängt. Für solche Fälle bietet WPF das Attribut **KeyboardNavigation.TabIndex** an, dass Sie für alle relevanten Steuerelemente hinzufügen:

```
<Window x:Class="TippsControls.Aktivierreihenfolge" ...>
    <Grid>
        <Label x:Name="label1" .../>
        <TextBox x:Name="txt1" ...
            KeyboardNavigation.TabIndex="1" />
        <Label x:Name="label2" .../>
        <TextBox x:Name="txt2" ...
            KeyboardNavigation.TabIndex="2"/>
        ...
    </Grid>
</Window>
```

Soll ein Steuerelement gar nicht in die Aktivierreihenfolge einbezogen werden, stellen Sie das Attribut **IsTabStop** auf den Wert **False** ein:

```
<TextBox x:Name="txt3" ... IsTabStop="False" />
```

Der Standardwert dieses Attributs lautet **True**.

Fenster zentrieren

Ohne weiteres Zutun öffnen sich die Fenster einer WPF-Anwendung irgendwo, zum Beispiel im linken, oberen Bereich des Bildschirms. Wenn Sie möchten, dass sich Fenster genau in der Mitte des Bildschirms öffnen, können Sie dies mit einer einfachen Einstellung erreichen. Dazu legen Sie im XAML-Code einfach den Wert des Attributs **WindowStartupLocation** für das **Window**-Element auf **CenterScreen** fest:

```
<Window ... Title="FensterZentrieren"
    Height="150" Width="300"
    WindowStartupLocation="CenterScreen">
```

Wenn Sie dies auch noch für ein vom Hauptfenster geöffnetes Fenster erledigen, wird dieses zentriert über dem anderen Fenster geöffnet (siehe Bild 2).

Das Attribut **WindowStartupLocation** hat drei mögliche Werte:

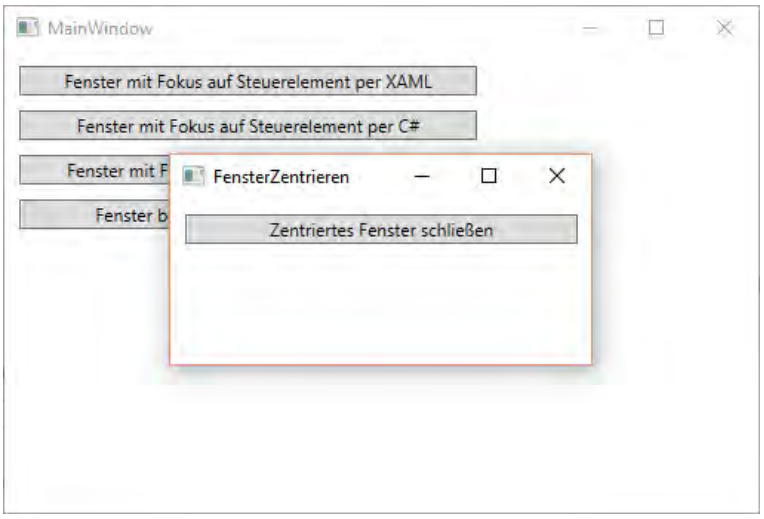


Bild 2: Ein zentriert geöffnetes Fenster

- **CenterScreen**: Fenster wird zentriert bezogen auf den Bildschirm angezeigt.
- **CenterOwner**: Fenster wird zentriert bezogen auf das aufrufende Fenster geöffnet.
- **Manual**: Fenster wird entsprechend der Werte für **Left** und **Top** geöffnet.

Fenster relativ zum Hauptfenster öffnen

Unter Access und VBA war das Öffnen eines Formulars relativ zur Position des aufrufenden Formulars erst nachträglich durch den Einsatz der Methode **DoCmd.MoveSize** möglich. Unter C# und WPF gelingt dies wesentlich einfacher. Das liegt daran, dass wir hier im Gegensatz zur unter Access üblichen Vorgehensweise mit der **DoCmd.OpenForm**-Methode das zu öffnende Fenster als Objekt betrachten und dieses erst deklarieren und initialisieren und dann erst öffnen. Unter Access war dies zwar auch mit Formularen möglich, aber wenig verbreitet.

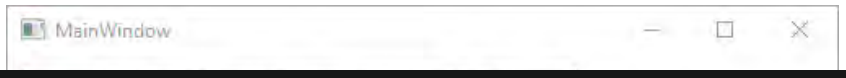
Dadurch, dass wir ein Fenster unter C# zunächst deklarieren und initialisieren, können wir über die Objektvariable auf seine Eigenschaften zugreifen und dementsprechend auch direkt etwa seine Position oder Größe einstellen. Hinzu kommt, das Hauptfenster einer WPF-Anwendung sowie die davon

```
    FensterMitRelativerPosition wnd =
        new FensterMitRelativerPosition();
    wnd.Left = this.Left + 200;
    wnd.Top = this.Top + 100;
    wnd.ShowDialog();
}
```

Die Methode deklariert und intialisiert ein Objekt auf Basis der Fensterklasse **FensterMitRelativerPosition**, dass wir zuvor als einfaches, leeres Fenster erstellen, und speichert den Objektverweis in der Variablen **wnd**. Dann weist es den beiden Eigenschaften **Left** und **Top** jeweils einen Wert zu, der um **200** größer als die entsprechenden Eigenschaften des aufrufenden Fensters sind. Danach öffnet die Methode das Fenster mit der **ShowDialog**-Methode (**Show** würde es an dieser Stelle auch tun). Das Ergebnis sehen Sie in Bild 3.

Eine zweite Variante unterscheidet sich in der Art der Zuweisung der Werte zu den Eigenschaften **Top** und **Left**. Diese hinterlegen wir für die Schaltfläche **btnFensterMitPositionRelativ2**:

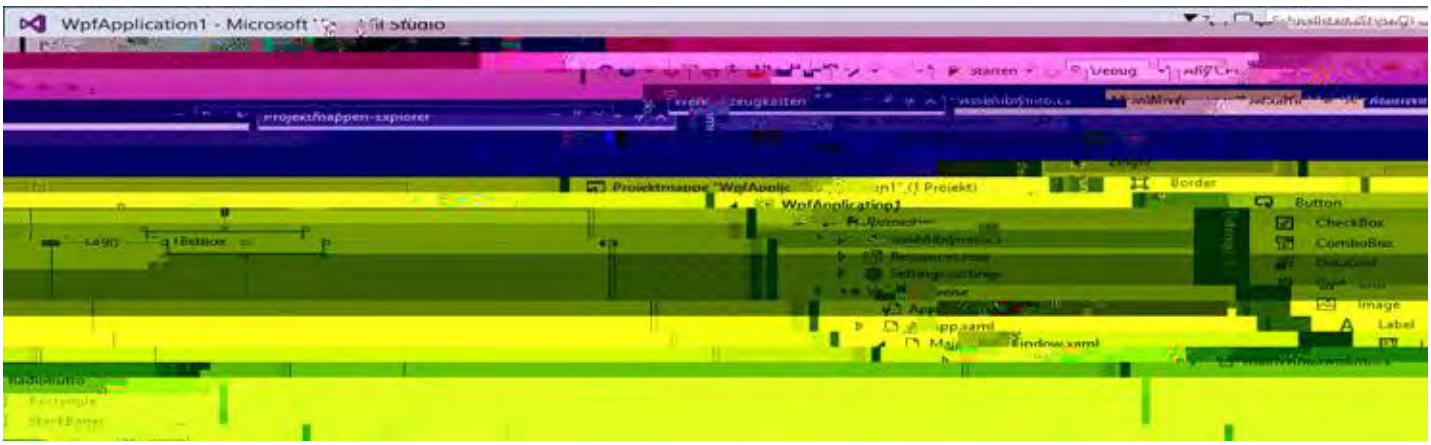
```
private void btnFensterMitPositionRelativ2_Click(
        object sender, RoutedEventArgs e) {
```



AUSGABE 03/2016

DATENBANK ENTWICKLER

MAGAZIN FÜR DIE DATENBANKENTWICKLUNG MIT VISUAL STUDIO FÜR DESKTOP, WEB UND CO.



TOP-THEMEN:

C#-GRUNDLAGEN	Von VBA zu C#: Fehlerbehandlung	SEITE 3
PROGRAMMIEREN	Objektorientierung: Interfaces	SEITE 19
INTERAKTIV	C#-DLL für COM/VBA erstellen	SEITE 24
WPF-BASICS	Datenbindung an einfache Objekte	SEITE 36
DATENZUGRIFF	Entity-Framework-Grundlagen	SEITE 51

Von VBA zu C#: Fehlerbehandlung

Nur wenige Access-Programmierer statten ihre Anwendungen mit einer ordentlichen Fehlerbehandlung aus. Für viele ist dies ein leidiges Thema. Die Fehlerbehandlung sorgt im Optimalfall sowohl unter VBA als auch unter C# dafür, dass eine Anwendung stabiler wird und nach dem Auftreten von Laufzeitfehlern nicht unerwartet reagiert oder sogar abstürzt. Während es unter VBA nur wenige Konstrukte gibt, um eine Fehlerbehandlung zu implementieren, bietet C# schon eine Menge mehr. Dieser Artikel liefert eine Einführung und zeigt, wie Sie die von VBA gewohnten Techniken unter C# einsetzen.

Die minimale Fehlerbehandlung unter VBA sieht so aus, dass Sie vor fehlerträchtigen Anweisungen die eingebaute Fehlerbehandlung mit der folgenden Programmzeile deaktivieren:

```
On Error Resume Next
```

Alle folgenden Fehler in der aktuellen Routine und in solchen, die von dieser Routine aufgerufen werden, werden nicht behandelt. Dadurch gibt es zwar immerhin keine Fehlermeldung und es werden keine Variablen geleert, was beim Auftreten unbehandelter Fehler auftreten kann. Allerdings erledigen fehlerhafte Zeilen ihre Aufgabe nicht, was zu Folgefehlern (auch logischen Fehlern) in den folgenden Anweisungen führen kann.

Spätestens mit dem Ende der Routine endet die Deaktivierung der eingebauten Fehlerbehandlung. Vorher erhalten Sie dies mit der folgenden Anweisung:

```
On Error Goto 0
```

Dadurch reagiert VBA wieder wie gewohnt mit entsprechenden Fehlermeldungen auf Fehler.

Dazwischen haben Sie Gelegenheit, benutzerdefiniert auf die auftretenden Fehler zu reagieren – beispielsweise, indem Sie den Wert der Eigenschaft **Number** des **Err**-Objekts auslesen und für die interessanten Werte entsprechende Fehlerbehandlungen hinzufügen.

Dies kann beispielsweise wie folgt aussehen:

```
On Error Resume Next
Debug.Print 1/0
Select Case Err.Number
    Case 11
        MsgBox "Fehler: Teilen durch 0."
End Select
On Error Goto 0
```

Unter C# gibt es zur Fehlerbehandlung andere Möglichkeiten, die Sie in den folgenden Abschnitten kennen lernen.

Warum überhaupt eine Fehlerbehandlung?

Um auf Nummer sicher zu gehen, wollen wir zuvor noch einmal kurz auf die Gründe für die Programmierung einer benutzerdefinierten Fehlerbehandlung eingehen. Wenn Sie

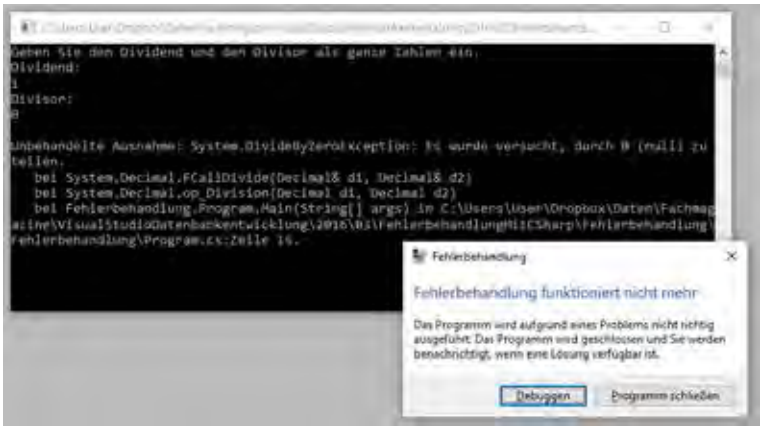


Bild 1: Unbehandelte Ausnahme bei einer Konsolenanwendung

Code programmieren, der fehlerhafte Eingaben zulässt und keine entsprechende benutzerdefinierte Fehlerbehandlung aufweist, erhalten Sie beispielsweise bei einer Konsolenanwendung eine Meldung plus Fehlerbehandlungsfenster wie in Bild 1.

Wenn Sie einen solchen Fehler in einer WPF-Anwendung auslösen, erhalten Sie noch nicht einmal einen kleinen Hinweis auf den Fehler im Code, der zu dieser Ausnahme führte (siehe Bild 2).

Noch übler wird es, wenn nicht Sie selbst die Software bedienen, sondern ein Kunde: Erstens macht dies nie einen guten Eindruck, zweitens kann der Kunde auf Basis der hier gelieferten Fehlerinformationen kaum helfen, den Fehler zu finden.

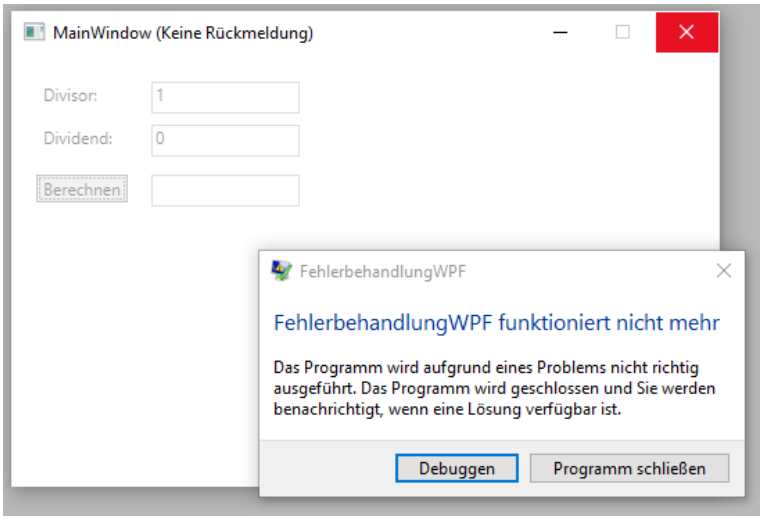


Bild 2: Unbehandelte Ausnahme bei einer Windows-Anwendung

Also sollten Sie dafür sorgen, dass der Kunde einen aussagekräftigen Text als Fehlermeldung erhält, der ihn beispielsweise bei einem Eingabefehler auf die Ursache des Fehlers hinweist oder aber ihm die Informationen liefert, die er zur Behebung des Problems an den Entwickler der Software weitergeben kann.

Der wichtigste Punkt beim Implementieren einer benutzerdefinierten Fehlerbehandlung etwa unter VBA ist jedoch die stabile Fortsetzung der Anwendung: Wenn ein Fehler aufgetreten ist, beispielsweise durch eine Fehleingabe oder einen fehlerhaften Zugriff auf ein Element, dann sollte die Anwendung nach der Ausgabe der Fehlermeldung fortgesetzt werden können und nicht einfach abbrechen. Dabei ist außerdem sicherzustellen, dass die Funktion und der stabile Zustand der Anwendung durch den Fehler nicht beeinträchtigt werden. Bei VBA war das mitunter ganz einfach deshalb nicht der Fall, weil durch unbehandelte Fehler Variableninhalte gelöscht oder Objekte zerstört wurden. Wurden die Zeilen, die den Fehler auslösten, hingegen per **On Error Resume Next** einer benutzerdefinierten Fehlerbehandlung zugeführt, statt einfach unbehandelte Fehler auszulösen, war zumindest schon einmal der Inhalt und der Zustand der Variablen sichergestellt.

Fehlerhafter Code

Schauen wir uns das Beispiel an, das zum Auslösen der Ausnahme der Konsolenanwendung führte. Den Code finden Sie in Listing 1. Die Zeile, die das Ergebnis der Division der Werte der Variablen **Dividend** und **Divisor** ermitteln soll, löst die Ausnahme aus, wenn der Divisor zuvor den Wert **0** zugewiesen bekommen hat – der Fehler tritt also durch eine Division durch **0** auf.

Wenn Sie die Anwendung debuggen, also diese von Visual Studio aus etwa mit

```
static void Main(string[] args) {
    Console.WriteLine("Geben Sie den Dividend und den Divisor als ganze Zahlen ein.");
    Console.WriteLine("Dividend:");
    decimal Dividend = Convert.ToDecimal(Console.ReadLine());
    Console.WriteLine("Divisor:");
    decimal Divisor = Convert.ToDecimal (Console.ReadLine());
    decimal Quotient = Dividend / Divisor;
    Console.WriteLine("Der Quotient lautet: {0}", Quotient);
    Console.ReadLine();
}
```

Listing 1: Methode, die einen Fehler auslöst, wenn als Divisor der Wert **0** angegeben wird

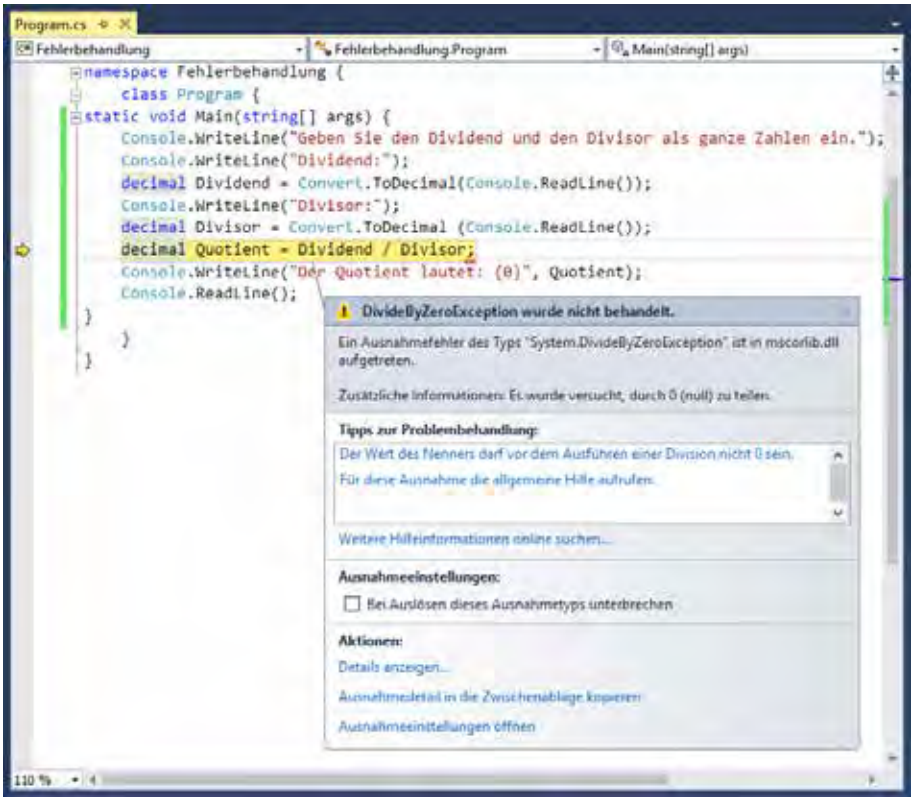


Bild 3: Auslösen einer Ausnahme unter C#

Bild 4: Details zu einer Ausnahme beim Debuggen in Visual Studio

der Taste **F5** starten, erhalten Sie einige Fehlerinformationen mehr. Außerdem markiert Visual Studio gleich die fehlerhafte Zeile (siehe Bild 3). Es gibt sogar noch Tipps zur Problembehandlung.

Wenn Sie möchten, erhalten Sie auch noch weitere Details, und zwar durch einen Mausklick auf den Link **Details anzeigen ...**, der den Dialog aus Bild 4 öffnet.

Für Fehler wie diese gibt es einige Beispiele – die Eingabe unzulässiger Werte wie in diesem Beispiel, Zugriff auf nicht vorhandene Dateien et cetera.

try...catch statt On Error Resume Next

Unter C# können Sie Laufzeitfehler in sogenannten **try...catch**-Blöcken behandeln. Das sieht dann so aus, dass Sie den Code, der einen Fehler auslösen könnte, in den **try**-Block packen und den Code, der ausgelöst werden soll, wenn innerhalb des **try**-Blocks ein Fehler auftritt, in den **catch**-Block.

In unserem Fall wollen wir die Zeile, welche die Division durchführt, in den **try**-Block überführen. Gleichzeitig fügen wir dort auch die Zeile ein, welche das Ergebnis aus der Variablen **Quotient** in der Konsole ausgibt – sonst ergibt dies einen Syntaxfehler, weil **Quotient** nicht in jedem Falle deklariert und initialisiert wird.

In den **catch**-Block schreiben wir eine Anweisung, welche einen Hinweis auf das Auftreten eines Fehlers in der Konsole ausgibt (siehe Listing 2). Wie Sie hier erkennen, wird die folgende Anweisung, die ja das Ergebnis der Berechnung ausgeben sollte, ignoriert – dies gilt grundsätzlich für alle Anweisungen, die der fehlerhaften Anweisung folgen.

Nun erhält der Benutzer zwar auch keine wesentlich aussagekräftigere Meldung, aber dafür wird das Programm auch nicht einfach abgebrochen. Außerdem haben wir ja auch

```
try {
    decimal Quotient = Dividend / Divisor;
    Console.WriteLine("Der Quotient lautet: {0}", Quotient);
}
catch {
    Console.WriteLine("Ups! Es ist ein Fehler beim Rechnen aufgetreten!");
}
```

Listing 2: Abfangen eines Fehlers per **try...catch**-Block

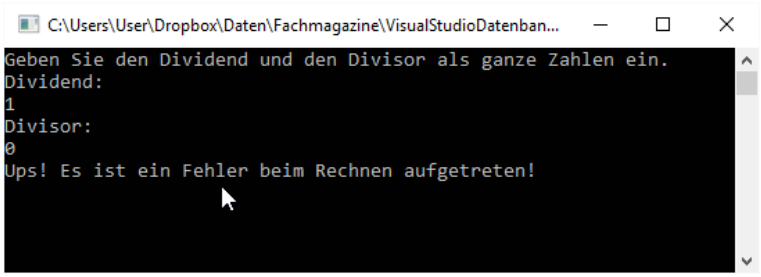


Bild 5: Benutzerdefinierte Fehlermeldung beim Auftreten der Ausnahme

noch gar nicht geprüft, um was für einen Fehler es sich handelt – dies prüfen wir in der folgenden Version.

Die Anwendung wird dann mit den Anweisungen fortgesetzt, die nach dem Ende des **catch**-Blocks folgen. Dies erkennen Sie in diesem Beispiel daran, dass das Betätigen der Eingabetaste das Programm beendet, weil die letzte **Console.ReadLine()**-Methode ausgeführt wird.

Exception statt Err

Unter C# heißt das Objekt, das die per Code auswertbaren Fehlerinformationen enthält, **Exception**. Es entspricht etwa dem Objekt **Err** unter VBA, das ja mit seinen Eigenschaften **Description** oder **Number** oft hilfreiche Informationen liefert.

Damit Sie dieses Objekt nutzen können, müssen Sie es für den **catch**-Zweig als Parameter hinzufügen, also wie in Bild 5 mit **catch (Exception e)**.

Die folgenden drei Zeilen des Beispiels geben verschiedene Informationen auf die Konsole aus, die Sie auch in Bild 6 sehen. Die erste ist der Inhalt der Eigenschaft **Message**. Die Eigenschaft **GetBaseException** liefert den Text, den auch ein unbehandelter Fehler auf die Konsole zaubert. Schließlich gibt uns die Eigenschaft **GetType** hilfreiche Informationen, wenn es darum geht, den Typ der Ausnahme zu ermitteln. In diesem Fall lautet der Typ **Divi-**

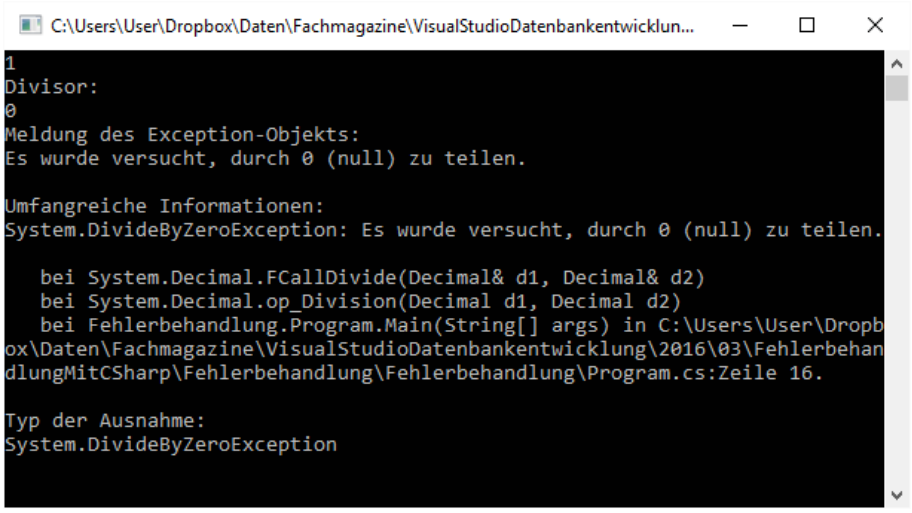


Bild 6: Erweiterte Fehlermeldung des **Exception**-Objekts

```
catch (Exception e) {
    Console.WriteLine("Meldung des Exception-Objekts:\n{0}\n", e.Message);
    Console.WriteLine("Umfangreiche Informationen:\n{0}\n", e.GetBaseException().ToString());
    Console.WriteLine("Typ der Ausnahme:\n{0}", e.GetType().ToString());
}
```

Listing 3: Ausstatten des **catch**-Block mit einigen weiteren Informationen

deByZeroException. Damit können wir später gezielt auf bestimmte Fehler reagieren.

Allgemeine Exception

Was wir im vorherigen Beispiel getan haben, war die Behandlung einer allgemeinen Exception. Die Klasse **Exception** ist quasi die Mutter aller Fehlerklassen. Es gibt eine ganze Reihe von Fehlerklassen, die von dieser Klasse abgeleitet sind.

Wenn Sie im **catch**-Zweig einer Ausnahmebehandlung in Klammern ein Objekt des Typs **Exception** übergeben, fangen Sie damit alle möglichen Ausnahmen ab. Dies entspricht etwa dem **Case Else**-Zweig in einer VBA-Fehlerbehandlung wie der folgenden:

```
Select Case Err.Number
    Case 11 'Geteilt durch Null
        'Fehler behandeln
    Case 0 'kein Fehler, nichts ist zu tun
    Case Else
        'Alle übrigen Fehlernummern behandeln
End Select
```

Nun wollen wir noch wissen, wie wir einen speziellen Fehler abfangen, in diesem Fall die **DivideByZeroException**. Wenn Sie nur diesen einen Fehler abfangen möchten, reicht die folgende Variante aus:

```
try {
    decimal Quotient = Dividend / Divisor;
    Console.WriteLine("Der Quotient lautet: {0}", Quotient);
}
catch (DivideByZeroException e) {
    Console.WriteLine("Der Dividend darf nicht 0 sein.");
}
```

Der **catch**-Zweig wird in diesem Fall nur angesteuert, wenn eine **DivideByZero**-Ausnahme ausgelöst wird. Im Falle eines jeden anderen Fehlers tritt eine unbehandelte Ausnahme auf.

Um einen weiteren, andersartigen Fehler auszulösen und behandeln zu können, ziehen wir die übrigen Anweisungen der Methode wie in Listing 4 ebenfalls in den **try**-Block hinein. Außerdem fügen wir neben dem **catch**-Block, der sich um den Fehler kümmert, der beim Teilen durch **0** auftritt, einen

```
public static void Andere_Exception() {
    try {
        Console.WriteLine("Geben Sie den Dividend und den Divisor als ganze Zahlen ein.");
        Console.WriteLine("Dividend:");
        decimal Dividend = Convert.ToDecimal(Console.ReadLine());
        Console.WriteLine("Divisor:");
        decimal Divisor = Convert.ToDecimal(Console.ReadLine());
        decimal Quotient = Dividend / Divisor;
        Console.WriteLine("Der Quotient lautet: {0}", Quotient);
    }
    catch (DivideByZeroException e) {
        Console.WriteLine("Der Dividend darf nicht 0 sein.");
    }
    catch (Exception e) {
        Console.WriteLine("Es ist ein Fehler aufgetreten: {0}", e.GetType().ToString());
    }
    Console.ReadLine();
}
```

Listing 4: Abfangen eines speziellen und eines allgemeinen Fehlers per **try...catch**-Block

weiteren **catch**-Block hinzu. Dieser soll wieder alle übrigen Fehler abfangen.

Wenn wir nun beispielsweise für die erste **Console.ReadLine**-Anweisung keine Zahl, sondern eine Zeichenkette eingeben, erhalten wir einen weiteren Fehler. Dieser wird ausgelöst, da wir das Ergebnis der Eingabe direkt in einen Wert des Datentyps **decimal** umwandeln wollen. Das gelingt mit einer Zeichenkette natürlich nicht, also liefert dies eine weitere Ausnahme – wie Bild 7 zeigt.

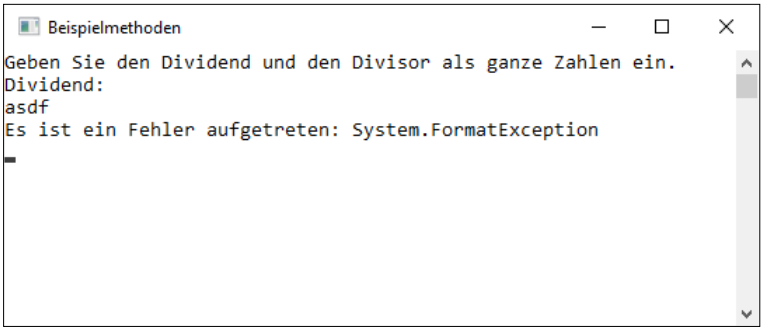


Bild 7: Fehler durch die Eingabe von Daten im falschen Format

Nachdem wir wissen, dass dies die Ausnahme **FormatException** auslöst, fügen wir auch diese als weiteren **catch**-Block zur Fehlerbehandlung hinzu:

```
try {
    ...
}
catch (DivideByZeroException e) {
    Console.WriteLine("Der Dividend darf nicht 0 sein.");
}
catch (FormatException e) {
    Console.WriteLine("Bitte geben Sie eine ganze Zahl ein.");
}
catch (Exception e) {
    Console.WriteLine("Es ist ein Fehler aufgetreten:
        {0}", e.GetType().ToString());
}
```

Auf ähnliche Weise legen wir für alle Fehler, die auftreten können, eine Ausnahmebehandlung an (zumindest für die, die wir bereits erkennen können – der Benutzer wird sicher noch weitere Schwachstellen aufdecken ...).

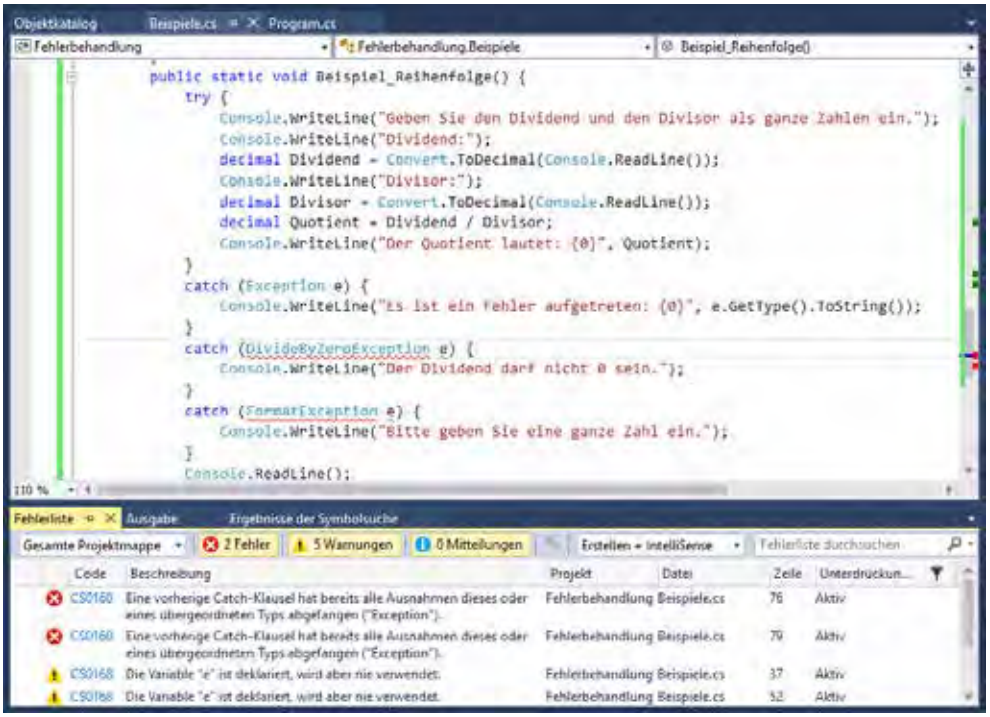


Bild 8: Exceptions müssen in der Hierarchie von unten nach oben abgearbeitet werden.

Die Reihenfolge der **catch**-Blöcke spielt natürlich eine übergeordnete Rolle, gerade wenn Sie eine allgemeine Ausnahmebehandlung (mit **catch (Exception e)**) und speziellere Ausnahmebehandlungen (wie mit **catch (DivideByZeroException)**) implementieren. Die catch-Blöcke werden nämlich immer in der angegebenen Reihenfolge abgearbeitet. Wenn Sie also direkt in der ersten

catch-Anweisung auf die allgemeine Ausnahme **Exception** prüfen, wird diese bei jeder denkbaren Ausnahme angesteuert, da ja alle anderen Ausnahmen von **Exception** erben. Aber Visual Studio beugt dem direkt vor, denn untergeordnete Ausnahmen müssen immer vor übergeordneten Ausnahmen abgearbeitet werden. Wenn wir versuchen, die allgemeine **Exception**-Ausnahme vor der **DivideByZeroException**-Ausnahme zu behandeln, erhalten wir einen Kompilierfehler (siehe Bild 8).

Hierarchie der Exceptions

Damit kommen wir zurück zur Mutter aller Fehlerklassen und den untergeordneten Elementen. Woher erfahren wir, in welcher Hierarchie-Ebene sich eine **...Exception**-Klasse befindet und welche die übergeordneten Klassen sind?

Dazu klicken Sie etwa im Code-Fenster mit der rechten Maustaste auf den Text **DivideByZeroException** und wählen dort den Eintrag **Definition einsehen** aus. Dies öffnet einen gelb hinterlegten Bereich, der die öffentliche Klasse beschreibt, die von der Klasse **ArithmeticException** abgeleitet ist (siehe Bild 9). Diese Abhängigkeit erkennen Sie an dem Doppelpunkt zwischen den beiden Klassennamen (**DivideByZeroException : ArithmeticException**).

Die Klasse **ArithmeticException** wird also vermutlich sämtliche Ausnahmen behandeln, die durch die Verwendung von Rechenoperatoren unter C# ausgelöst werden, wobei **DivideByZeroException** nur ein Spezialfall ist.

Es geht aber noch weiter: Wenn Sie in der Definition mit der rechten Maustaste auf den Eintrag **ArithmeticException** klicken und dann wiederum den Kontextmenü-Eintrag **Definition einsehen** auswählen, erhalten Sie die Definition der Klasse **ArithmeticException** und mit **SystemException** die übergeordnete Ausnahmeklasse. Dass die Hierarchie einigermaßen flach ist, erkennen Sie, wenn Sie auch noch die übergeordnete Klasse von **SystemException** ermitteln – hier landen Sie dann nämlich bei der Klasse **Exception**, die keine weitere übergeordnete Ausnahmeklasse besitzt.

Exception mit oder ohne Variable?

In allen bisherigen Beispielen haben wir das **Exception**-Objekt in einer Variablen namens **e** gespeichert, um innerhalb des **catch**-Blocks auf die Eigenschaften des Ausnahme-Objekts zugreifen zu können. Diese Variable können Sie auch weglassen, wenn Sie gar nicht auf die Eigenschaften zugreifen wollen. Meist ist dies nicht notwendig – im Falle der **DivideByZeroException** etwa gibt es in der betroffenen Methode ja nur eine Zeile, die diesen Fehler auslösen kann, und der Grund dafür ist dann auch offensichtlich. Sie können e also auch einfach weglassen:

```
...
catch (DivideByZeroException) {
    Console.WriteLine("Der Dividend darf nicht 0 sein.");
}
...
```

In diesem Fall erscheinen dann auch keine Warnungen wie **Die Variable "e" ist deklariert, wird aber nie verwendet.** mehr in der Fehlerliste von Visual Studio.

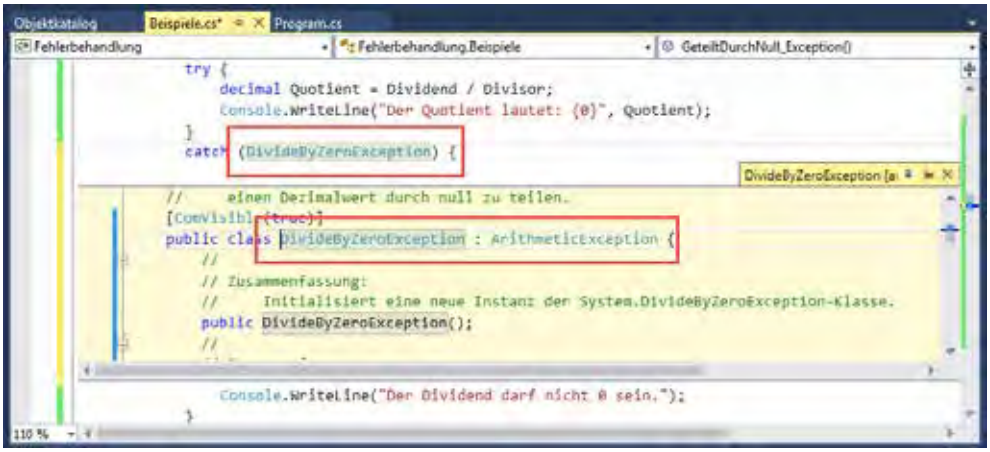


Bild 9: Die Definition einer Klasse liefert Hinweise auf die übergeordnete Klasse

Objektorientierte Programmierung: Interfaces

Interfaces beziehungsweise Schnittstellen bezeichnen in der Computerwelt eine Menge verschiedener Dinge. Es gibt eine Benutzerschnittstelle, Softwareschnittstellen, Maschinenschnittstellen und mehr. Uns interessiert das Interface, wie es in der objektorientierten Programmierung genutzt wird. Schnittstellen dienen dort als zuverlässige Definition der verfügbaren Methoden, Eigenschaften oder Ereignisse für eine oder mehrere Klassen, welche diese tatsächlich definieren.

Auf diese Weise kann man sich beim Programmieren »gegen« eine Schnittstelle darauf verlassen, dass die Klasse, welche die Schnittstelle implementiert, auch die dort definierten Methoden, Eigenschaften, Ereignisse und Indexer enthält (Indexer haben Sie im Rahmen dieses Magazins noch nicht kennen gelernt, daher gehen wir in diesem Artikel nicht auf diese Elemente ein).

Wozu aber sollte man überhaupt verschiedene Klassen programmieren, die alle die gleichen Methoden, Eigenschaften oder Ereignisse enthalten und somit die Schnittstelle implementieren? Ein Grund ist, dass Sie nur einmal lernen möchten, wie etwas funktioniert (die in der Schnittstelle definierten Elemente), die Implementierung jedoch je nach Anwendungsfall variieren kann.

Aus dem Leben gegriffen

Stellen Sie sich das Anziehen oder Lösen einer Schraube vor: Sie verwenden den für die Schraube passenden Schraubendreher, zum Beispiel einen Kreuzschlitz- oder einen Längsschlitz-Schraubendreher. Nun haben beide Schraubendreher vielleicht unterschiedliche Griffe, wodurch Sie jeweils anders greifen müssen. Nun stellen Sie sich einen Schraubendreher mit austauschbaren Bits vor: Die (Benutzer-)Schnittstelle ist der Griff, mit dem Sie den Schraubendreher nach links oder rechts drehen können, aber je nach Schraube verwenden Sie ein anderes Bit, um diese anzuziehen oder zu lösen. Sie kennen also die beiden Methoden der Schnittstelle, also nach links und nach rechts drehen, und können verschiedene Implementierungen nutzen – also etwa ein Bit für Kreuzschlitz- und ein anderes für Längsschlitz-Schrauben. Genau genommen ist auch die Schnittstelle zwischen dem Schraubendreher und dem Bit ein Interface – wir haben also gleich ein doppeltes Beispiel für ein Interface.

Beispiel: E-Mail versenden

Genau das Gleiche kann in der Software-Welt geschehen: Sie möchten beispielsweise eine E-Mail verschicken und dazu einem Objekt den Empfänger, den Betreff und den Inhalt übergeben (die Eigenschaften) und die Mail dann mit der **Senden**-Methode abschicken. Welcher Mechanismus dahinter steckt, soll für den Aufruf uninteressant sein – es könnte sich um die Programmierung von Outlook handeln, um die Mail mit diesem Programm zu versenden, aber vielleicht verwendet eine andere Implementierung nicht Outlook, sondern eine speziell für den Versand von E-Mails vorgesehene SMTP-Schnittstelle.

Beispiel: Daten abrufen

Oder, was in diesem Magazin auch bereits das eine oder andere Mal beschrieben wurde: Sie möchten ein Objekt mit Daten füllen, also etwa ein Kunden-Objekt mit den Eigenschaften **KundeID**, **Vorname** und **Nachname**. Wenn Sie mit der Programmierung beginnen, sollen die Daten aus einer Access-Datenbank kommen. Aber vielleicht wechseln Sie demnächst zu einer SQL Server-Datenbank? Oder lesen die Daten aus einer XML-Datei ein? Dann benötigen Sie in jedem Fall unterschiedliche Implementierungen der Methoden, da der Zugriff auf die Daten ja nicht auf die gleiche Art erfolgt.

Verschiedene Methoden oder verschiedene Implementierungen?

Natürlich könnten Sie, um beim Beispiel des Abrufs von Daten aus verschiedenartigen Datenquellen zu bleiben, für

jeden Zugriff eine andere Methode implementieren – also beispielsweise **HoleKundeAccess**, **HoleKundeSQLServer** oder **HoleKundeXML**. Dann müssten Sie allerdings auch irgendwo eine Bedingung unterbringen, die prüft, welche der Methoden zum Abrufen eines Kunden-Objekts aus einer der verschiedenen Datenquellen zum Einsatz kommen soll.

In der Objektorientierung soll dies jedoch anders ablaufen. Hier definieren Sie jeweils eine Klasse mit Methoden für den Zugriff auf die Daten der verschiedenen Datenquellen, also etwa Access, SQL Server oder XML. Dort sollen natürlich für jede Datenquellenart noch weitere Eigenschaften, Methoden oder auch Ereignisse definiert werden. Würden Sie diese alle in einer Klasse unterbringen, hätten Sie erstens eine recht unübersichtliche Anhäufung von Elementen. Außerdem gäbe es keine Prüfung, ob tatsächlich alle Methoden, Eigenschaften und Ereignisse für die verschiedenen Datenquellen definiert wurden.

Wenn Sie hingegen eine Schnittstelle beziehungsweise ein Interface definieren und für jede Datenquellenart eine eigene Klasse erstellen, welche die in der Interface-Klasse definierten Methoden, Eigenschaften und Ereignisse implementiert, erhalten Sie automatisch eine Prüfung, ob alle benötigten Elemente vorliegen. Dazu müssen Sie beim Programmieren der Klasse nur angeben, welche Schnittstelle diese implementiert.

Interfaces zur Einschränkung

Es gibt noch einen weiteren Grund, eine Schnittstelle für eine Klasse zu programmieren. Die Grundlagen dazu finden Sie im Artikel **C#-DLL für COM/VBA erstellen**. Wenn Sie eine DLL mit Visual Studio erstellen, wollen Sie nur genau die dafür vorgesehenen Eigenschaften, Methoden und Ereignisse bereitstellen. Ohne eine Schnittstelle ist dies jedoch nicht so einfach möglich, denn die Standardelemente eines Objekts werden ebenfalls angeboten. Mithilfe eines Interfaces können Sie die zu veröffentlichenden Elemente gezielt einschränken.

Einfaches Beispiel für ein Interface

Für ein Beispiel einer Schnittstelle wollen wir verschiedene Methoden und Eigenschaften nutzen. Diese definieren wir in der Interface-Klasse, die sich von einer normalen Klasse vor allem in drei Punkten unterscheidet:

- Sie verwendet das Schlüsselwort **Interface** statt **Class**.
- Sie definiert nur die Member der Klasse, aber enthält keinerlei Code.
- Ihr Name sollte mit einem großen **i** beginnen (**I**).

Interfaces können Sie als **internal** (Standard) oder **public** deklarieren. Ein mit **internal** deklariertes Interface ist innerhalb der Anwendung nutzbar, ein als **public** deklariertes auch von außen – also beispielsweise, wenn ein COM-Client auf eine entsprechend ausgestattete C#-DLL zugreift.

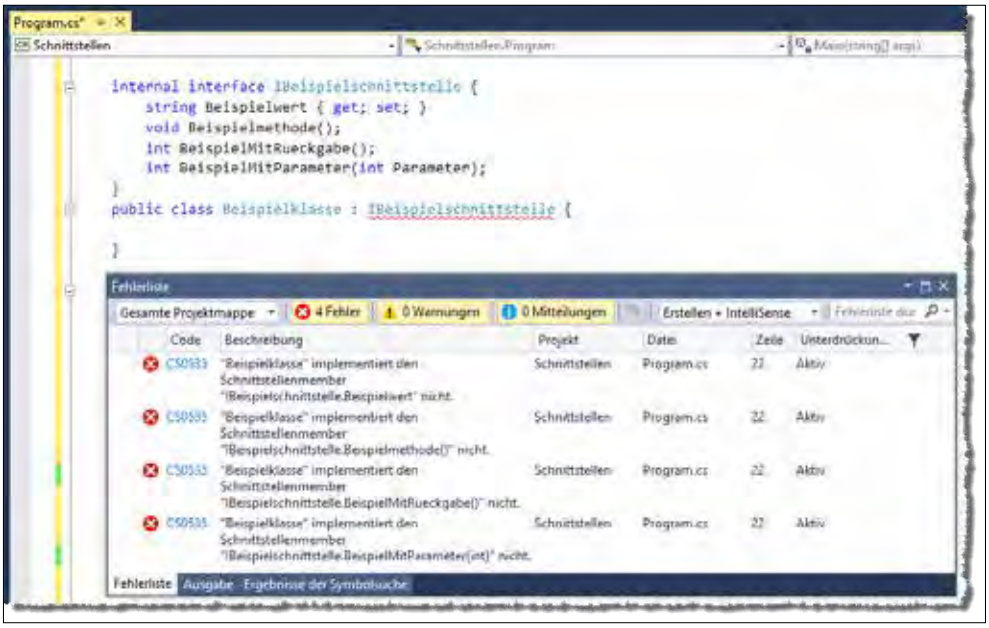


Bild 1: Fehlermeldungen für nicht implementierte Schnittstellenmember

C#-DLL für COM/VBA erstellen

Das .NET-Framework bietet im Vergleich zu VBA eine schier unendliche Menge nutzbarer Bibliotheken mit Objekten, Methoden und Eigenschaften für die verschiedensten Anwendungsfälle. Wer noch nicht komplett auf eine .NET-Anwendung umsteigen möchte, mag sich aber vielleicht die Möglichkeiten des Frameworks unter VBA erschließen. Dazu programmieren Sie eine DLL-Bibliothek, welche die benötigen .NET-Elemente enthält und für externe Anwendungen wie etwa eine Access-Datenbank bereithält. Der vorliegende Artikel erklärt, wie dies funktioniert.

Grundlagen

Wenn es darum geht, von einem VBA-Projekt etwa auf die Methoden einer mit C# programmierten DLL zuzugreifen, benötigen Sie eine Technologie namens COM (Component Object Model). Diese wurde von Microsoft bereits vor einer ganzen Weile eingeführt, nämlich in den frühen neunziger Jahren. Das Ziel war, dass Komponenten, die mit verschiedenen Programmiersprachen programmiert wurden, miteinander kommunizieren können.

COM liefert die Vorgaben für die Schnittstellen, die diese Interaktion ermöglichen. Damit nun auch noch Komponenten, die auf .NET basieren, mit COM-Objekten zusammenarbeiten, gibt es eine weitere Technologie, die sich COM Interop nennt. Diese erlaubt es beispielsweise, die in einer .NET-DLL definierten Objekte, Methoden und Eigenschaften auch für eine COM-Komponente zugänglich zu machen.

Eine .NET-Komponente ist beispielsweise eine DLL-Datei. Damit Sie von COM-Objekten auf diese DLL-Datei zugreifen können, müssen Sie zunächst eine COM-Type Library (**.tlb**) erstellen, welche die relevanten Informationen bereithält. Dies erledigen Sie mit einem Befehlszeilentool namens **REGASM.EXE** oder **TLBEXP.EXE**.

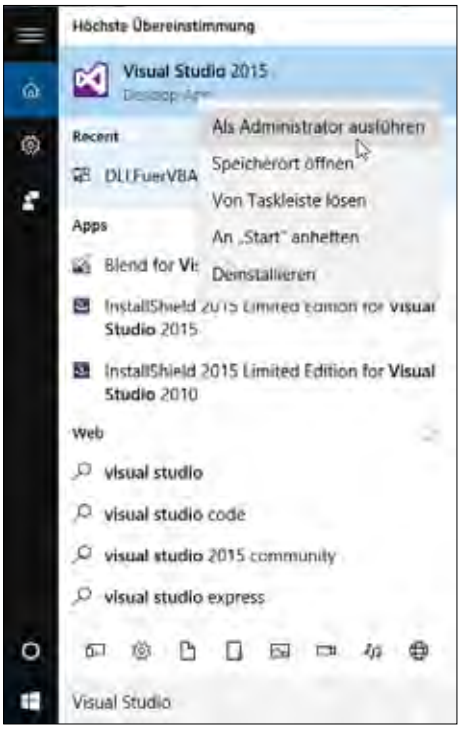


Bild 1: Visual Studio als Administrator starten

Wichtig: Admin-Modus!

Wenn Sie eine DLL erstellen und registrieren möchten, müssen Sie Visual Studio als Administrator ausführen. Dazu klicken Sie mit der rechten Maustaste auf den Visual Studio-Eintrag, der beispielsweise erscheint, wenn Sie im Suchfeld den Suchbegriff **Visual Studio** eingeben. Im folgenden Kontextmenü wählen Sie den Eintrag **Als Administrator ausführen** aus (siehe Bild 1). In diesem Modus können die notwendigen Registry-Einträge gesetzt werden.

Stolperfalle: DLL in Verwendung

Sollten Sie die DLL bereits einmal erstellt haben, wollen Sie diese natürlich im VBA-Projekt einer Access-Anwendung testen. Wenn Sie die DLL etwa per Verweis referenzieren und die Datenbank geöffnet ist, kann die referenzierte DLL nicht überschrieben werden, da diese schreibgeschützt ist. Sollten Sie das C#-Projekt mit der DLL zu diesem Zeitpunkt also in Visual Studio editieren und neu erstellen wollen, gelingt dies nicht, da die DLL ja nicht überschrieben werden kann. Dies erfahren Sie dann auch im Bereich **Fehlerliste** (siehe Bild 2).

Also, auch wenn es mit der Zeit nervt: Nicht vergessen, die Anwendung, mit der Sie die DLL testen, vor dem Neuerstellen der DLL zu schließen.

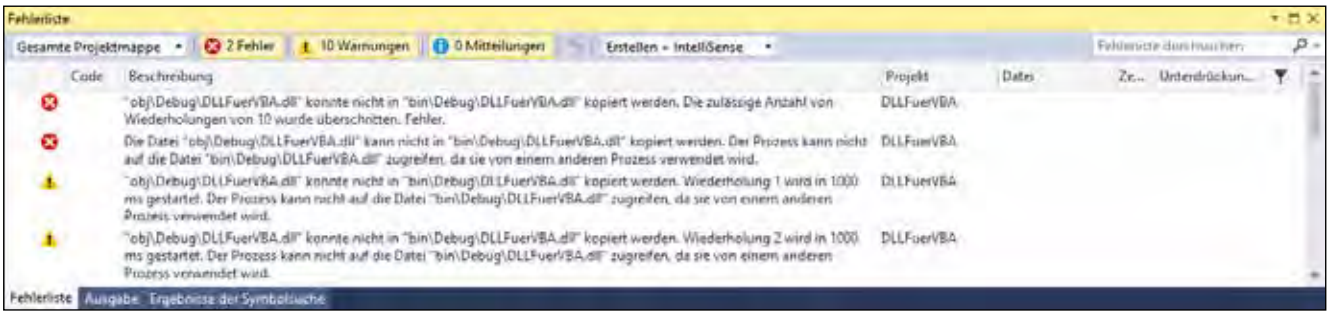


Bild 2: Fehler beim Versuch, eine aktuell in Verwendung befindliche DLL neu zu erstellen

Aber: Es gibt eine einfachere Methode, die wir weiter unten nach der Erstellung des ersten Beispiels erläutern.

DLL erstellen

Um eine DLL zu erstellen, deren Funktionen Sie später von einem VBA-Projekt nützen können, legen Sie in Visual Studio ein Projekt auf Basis der Vorlage **Klassenbibliothek** an (siehe Bild 3). Ändern Sie den Namen der automatisch hinzugefügten Klassendatei namens **Class1** in **LateBinding** um und erledigen Sie dies auch für den Klassennamen selbst.

Danach fügen Sie dem Klassenmodul den folgenden einfachen Code hinzu. Die Klasse deklariert zwei Variablen namens **zahl1** und **zahl2**, die über die Eigenschaften **Zahl1** und **Zahl2** gefüllt werden sollen. Die Methode **Produkt** soll dann das Produkt der beiden Zahlen zurückliefern:

```
namespace DLLFuerVBA {
    public class NoIntelliSense {
        int zahl1;
        int zahl2;
        public int Zahl1 {
            set { zahl1 = value; }
        }
        public int Zahl2 {
            set { zahl2 = value; }
        }
        public int Produkt() {
            return zahl1 * zahl2;
        }
    }
}
```

Dass wir die Klasse **NoIntelliSense** nennen, hat seinen Grund – mehr dazu weiter unten.

Der Clou ist nun, dass diese Klasse nicht innerhalb des C#-Projekts genutzt werden soll, sondern von einem VBA-Projekt in einer Access-Anwendung. Dazu sind zwei bereits durchgeführte und noch zwei weitere Schritte notwendig:

- Der erste bereits erledigte Schritt ist, dass die Klassendefinition als **Public** deklariert wird.
- Der zweite Schritt war, dass auch die Member der Klasse, die von außen erreicht werden sollen, das Schlüsselwort **Public** enthalten.
- Der dritte Schritt ist, dass Sie die Option **Für COM-Interop registrieren** aktivieren (siehe weiter unten).

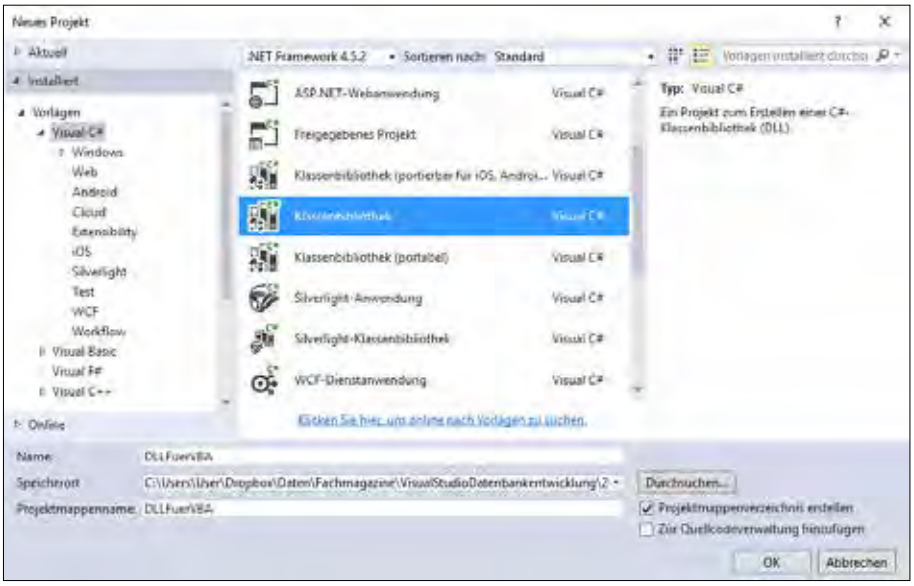


Bild 3: Hinzufügen eines Projekts, das wir als DLL verwenden können

- Der vierte Schritt ist das Aktivieren der Option **Assembly COM-sichtbar machen** (siehe ebenfalls weiter unten).

Für COM-Interop registrieren

Die Option **Für COM-Interop registrieren** finden Sie, wenn Sie im Projektmappen-Explorer mit der rechten Maustaste auf den Projektnamen klicken, den Eintrag **Eigenschaften** anklicken und im dann erscheinenden Fenster zum Bereich **Erstellen** wechseln. Unten unter **Ausgabe** finden Sie die gesuchte Option, die Sie per Klick auf das Kontrollkästchen aktivieren (siehe Bild 4). Wenn Sie diese Option nicht aktivieren, erstellt Visual Studio beim Erstellen des Projekts im Unterordner **\bin\debug** nur die übliche **.dll**-Datei plus einer **.pdb**-Datei. Damit Sie die DLL von einem VBA-Projekt aus nutzen können, müssen Sie diese jedoch entsprechend registrieren. Das können Sie auf zwei Arten erledigen – entweder mit einem Kommandozeilentool oder durch die Aktivierung der obigen Option vor dem Erstellen des Projekts. Dabei werden zwei Schritte durchgeführt: Visual Studio erstellt eine weitere Datei mit der Endung **.tlb**, welche die Definition der Schnittstelle enthält, die Sie später vom VBA-Projekt aus nutzen können. Außerdem legt Visual Studio auch gleich einige Einträge in der Registry an, damit Sie die Bibliothek über den Verweise-Dialog des VBA-Editors auswählen können.

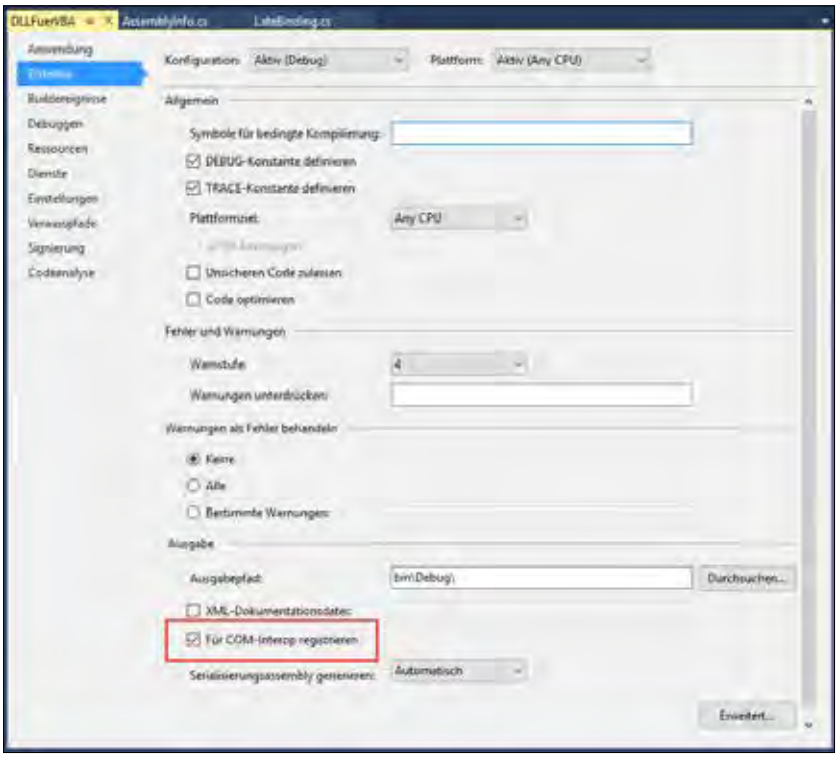


Bild 4: Einstellen der Option **Für COM-Interop registrieren**

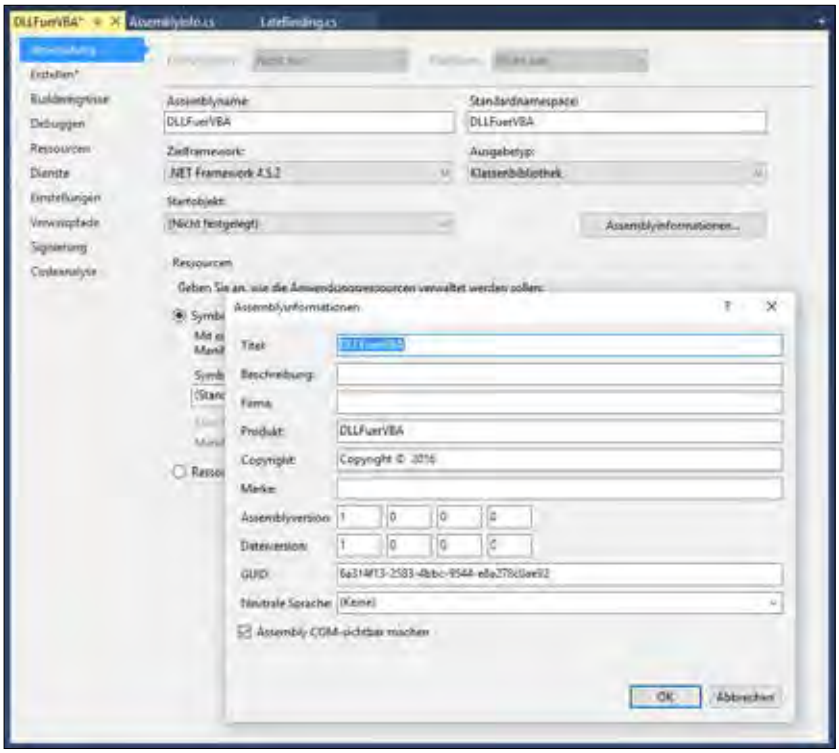


Bild 5: Aktivieren der Option **Assembly COM-sichtbar machen**

Assembly COM-sichtbar machen

Der zweite notwendige Schritt ist wieder mit dem Setzen einer einfachen Option erledigt. Diese ist jedoch etwas besser versteckt. Wechseln Sie in den Projekteigenschaften zum Bereich **Anwendungen** und klicken Sie dort auf die Schaltfläche **Assemblyinformationen**. Dies öffnet den Dialog **Assemblyinformationen**, der wie in Bild 5 aussieht. Hier aktivieren Sie

im unteren Bereich die Option **Assembly COM-sichtbar machen** und schließen den Dialog wieder.

Einfaches Projekt erstellen

Wenn Sie nun das Projekt mit dem obigen Code erstellen, legt Visual Studio einige

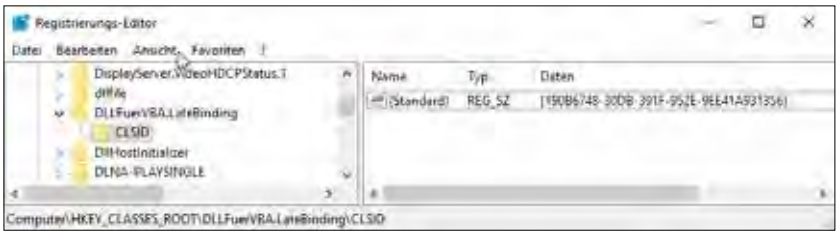


Bild 6: Erster Eintrag in der Registry

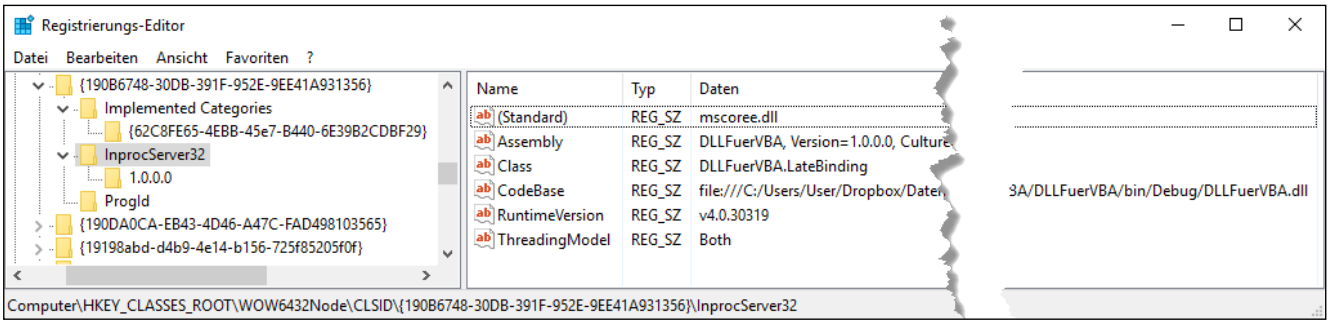


Bild 7: Details zur Registrierung der DLL

Einträge in der Registry des aktuellen Rechners an. Der erste Eintrag, der für uns wichtig ist, befindet sich in der Registry, die Sie mit dem Befehl **RegEdit** öffnen, unter dem Pfad **HKEY_CLASSES_ROOT\DLLFuerVBA.LateBinding**. Die wichtige Information dieses Pfades befindet sich im Schlüssel CLSID. Dort finden Sie nämlich eine GUID, die auf den Schlüssel mit weiteren Informationen hinweist (siehe Bild 6).

Diesen finden Sie in unserem Beispiel unter folgendem Pfad (die GUID kann variieren): **HKEY_CLASSES_ROOT\WOW6432Node\CLSID\{190B6748-30DB-391F-952E-9EE41A931356}**. Unterhalb dieses Schlüssels finden Sie den Schlüssel **InprocServer32**, der einige Informationen liefert (siehe Bild 7).

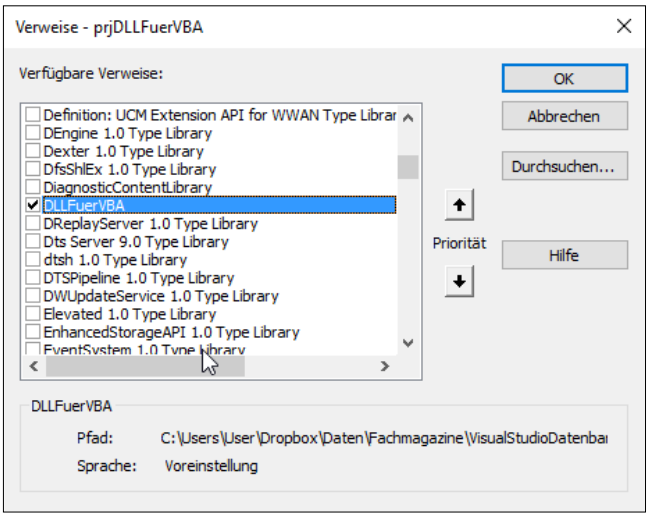


Bild 8: Der Verweis auf die Objektbibliothek lässt sich bereits setzen.

Objektbibliothek verfügbar

Zu diesem Zeitpunkt können Sie auch bereits einen Verweis vom VBA-Projekt einer Access-Datenbank auf die DLL beziehungsweise die erstellte **.tlb**-Datei setzen – dazu öffnen Sie einfach den **Verweise**-Dialog (VBA-Editor, Menüeintrag **Extras|Verweise**) und suchen den Eintrag **DLLFuerVBA** (siehe Bild 8).

Wenn wir uns das Objekt nun allerdings im Objektkatalog ansehen, finden wir dort noch keine Eigenschaften, Methoden oder Ereignisse (siehe Bild 9).

Nun erfahren Sie, warum wir unsere erste Klasse **NoIntelliSense** genannt haben: Wie zu erwarten, finden sich nicht nur keine Elemente im Objektkatalog, sondern auch der Versuch, per IntelliSense auf die Eigenschaften eines Objekts auf Basis der Klasse **DLLFuerVBA.NoIntelliSense** zuzugreifen, schlägt fehl. Immerhin können wir die Eigenschaften **Zahl1** und **Zahl2** sowie die Methode **Produkt** überhaupt nutzen:

```
Public Sub Test_EarlyBinding()
    Dim obj As DLLFuerVBA.NoIntelliSense
    Set obj = New DLLFuerVBA.NoIntelliSense
    With obj
        .Zahl1 = 2
        .Zahl2 = 3
        Debug.Print .Produkt
    End With
End Sub
```

Diese VBA-Prozedur gibt den erwarteten Wert **6** im Direktfenster aus. Nun fehlt uns allerdings noch die IntelliSense-Unterstützung und auch die Anzeige der Elemente des Objekts im Objektkatalog. Darum kümmern wir uns nach einem kleinen Ausflug in die Grundlagen für die Beziehung zwischen .NET und COM.

Das Bindeglied zwischen .NET und COM-Clients

Wenn Sie eine .NET-Anwendung erstellen möchten, auf deren Objekte Sie von einem COM-Client wie einer Access-Anwendung zugreifen wollen, benötigen Sie ein Bindeglied zwischen dem Managed Code von .NET und dem Unmanaged COM -Client.

Dieses heißt **COM Callable Wrapper** (**CCW**) und kann als eine Art Proxy angesehen werden. Dieser soll dafür sorgen, dass Sie mit einem COM-Client auf eine .NET-Klasse (oder auch eine **.exe**-Datei) wie auf ein anderes COM-Objekt zugreifen können. Dabei kümmert es sich um die korrekte Zuordnung der Datentypen, die sich ja auf beiden Seiten unterscheiden können, und stellt eine COM-Schnittstelle zur Verfügung. Außerdem organisiert es im Hintergrund die Mechanismen, die dafür sorgen, dass ein .NET-Objekt nach erfolgtem Zugriff gegebenenfalls beendet und der Speicher freigegeben wird.

Schnittstelle zwischen COM und .NET

Damit unser COM-Client, also beispielsweise das VBA-Projekt, mit der .NET-DLL kommunizieren kann, benötigen wir eine Schnittstelle. Diese kommt in Form der **.tlb**-Datei beziehungsweise als Type Library. Die **.tlb**-Datei wird beim

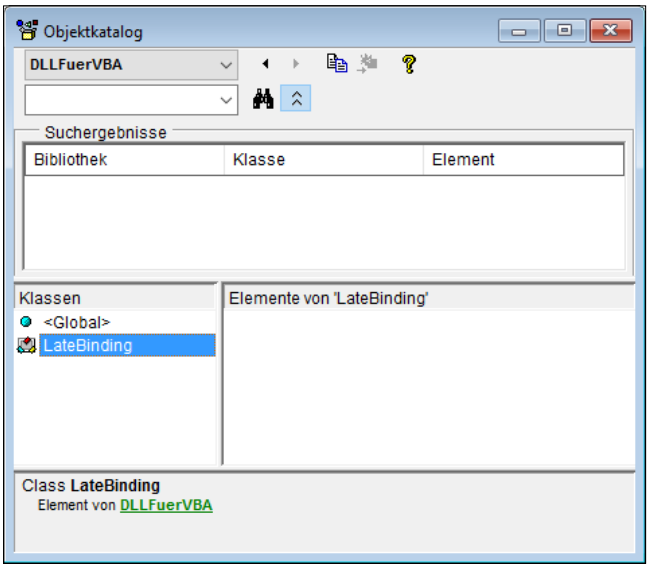


Bild 9: Der Objektkatalog offenbart keinerlei Eigenschaften, Methoden oder Ereignisse.

Erstellen des Projekts in Visual Studio automatisch generiert, wenn Sie die obigen Optionen **Für COM-Interop registrieren** und **Assembly COM-sichtbar machen** aktiviert haben. Alternativ können Sie die Erstellung dieser **.tlb**-Datei sowie die Registrierung mit einem Befehlszeilentool wie **tlbexp.exe** oder **regasm.exe** durchführen. Die Erstellung der Type Library basiert dabei auf dem Inhalt des Interface Definition Language-Codes in der von Visual Studio erstellten DLL.

IDL-Code ansehen

Wenn Sie genau wissen möchten, wie sich die Optionen und auch die weiter unten vorgestellten verschiedenen Deklarationen von Klassen und Membern auswirken, können Sie sich den Inhalt einer COM-DLL ansehen. Dazu verwenden Sie das Tool **OleView.exe**, das Sie etwa unter folgendem Link bei Microsoft herunterladen können: **http://download.microsoft.com/download/win2000platform/oleview/1.00.0.1/NT5/EN-US/oleview_setup.exe**

Nach der Installation finden Sie das Tool im Verzeichnis **C:\Program Files (x86)\Resource Kit** unter **oleview.exe**. Gegebenenfalls müssen Sie noch eine weitere DLL namens **iviewers.dll** herunterladen und im gleichen Verzeichnis speichern. Diese DLL findet sich an verschiedenen Stellen im Internet, hier empfiehlt sich eine Google-Suche.

Starten Sie **OleView.exe**, erhalten Sie das Fenster aus Bild 10.

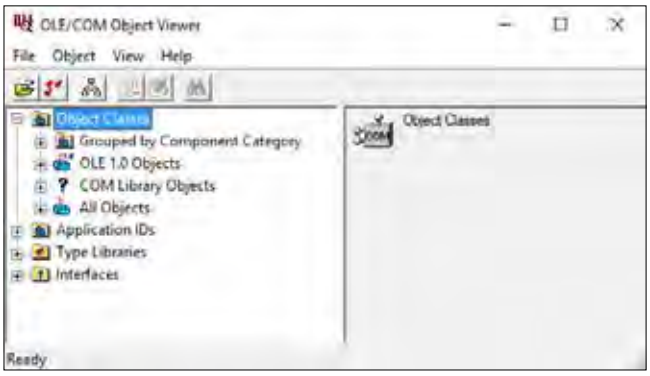


Bild 10: Das Tool **OleView.exe**

Der für uns interessante Bereich heißt **All Objects**. Erweitern Sie diesen, klicken auf das erste Objekt und geben dann die ersten Zeichen des Projektnamens ein, landen Sie schnell beim gesuchten Eintrag (siehe Bild 11). Was Sie hier sehen, sind die verschiedenen Interfaces für das Objekt. Auch wenn wir im C#-Projekt selbst kein Interface, also keine Schnittstelle definiert haben, wird eine Standard-Schnittstelle festgelegt – in diesem Fall **_NoIntelliSense**. Nun wollen wir uns den IDL-Code für diese Schnittstelle ansehen. Dazu klicken Sie mit der rechten Maustaste auf den Eintrag **_NoIntelliSense** und wählen aus dem Kontextmenü den Eintrag **View** aus. Im folgenden Dialog klicken Sie auf **View Type Info**.

Nun öffnet sich das Fenster **ITypeInfo Viewer** (siehe Bild 12) und zeigt die IDL-Definition für die Standard-Schnittstelle unserer Klasse an. Der IDL-Code sieht überschaubar aus: Hier findet sich kein Hinweis auf unsere beiden Eigenschaften **Zahl1** und **Zahl2** und auch nicht auf die Methode **Produkt**.

Keine Schnittstelle?

Warum ist dies der Fall und warum werden unsere Methoden und Eigenschaften nicht unter VBA angezeigt? Der Grund ist die Wahl der Schnittstelle – beziehungsweise die Wahl keiner Schnittstelle. Sie könnten die Klasse mit einer Klassenschnittstelle versehen, aber im aktuellen Zustand ist dies nicht der Fall. Um dies zu ändern, erstellen wir eine neue Klasse, diesmal namens **EarlyBinding**, aber mit dem gleichen Inhalt. Wir fügen der Klasse nur zwei Zeilen hinzu:

```
namespace DLLFuerVBA {
    using System.Runtime.InteropServices;
    [ClassInterface(ClassInterfaceType.AutoDual)]
    public class EarlyBinding {
        int zahl1;
        int zahl2;
        public int Zahl1 {
            set { zahl1 = value; }
        }
    ...
    }
}
```

Die Zeile mit der **using**-Direktive fügt den Namespace **System.Runtime.InteropServices** zum Namespace **DLLFu-**

Bild 11: Die Standard-Schnittstelle unserer DLL

WPF-Datenbindung: Einfache Objekte

Wenn Sie WPF-Anwendungen erstellen wollen, wollen Sie die Daten der Anwendung in entsprechenden Fenstern darstellen. Ein Weg zu diesem Ziel ist das Bereitstellen der Daten in Form von Objekten auf Basis von Klassen, welche die Eigenschaften des jeweiligen Objekts beschreiben. Dieser Artikel zeigt, welche Schritte nötig sind, um ein Objekt auf Basis einer einfachen Klasse zu erstellen und seine Daten in den Steuerelementen eines WPF-Fensters anzuzeigen.

WPF-Anwendung erstellen

Da wir nicht nur ein Objekt erstellen, sondern auch seine Daten in einem WPF-Fenster abbilden möchten, benötigen wir ein Projekt auf Basis der Vorlage **Visual C#\WPF-Anwendung**, die wir im Falle des Beispielprojekts **WPFDatenbindung** benannt haben.

Beispielklasse für einen Kunden erstellen

Gleich danach erstellen wir die Beispielklasse, welche die Definition für die Kunden-Objekte enthält. Diese fügen Sie beispielsweise über den Kontextmenü-Eintrag **Hinzufügen\Klasse...** des Projektnamens im Projektmappen-Explorer hinzu. Legen Sie den Namen **Kunde** für die Klasse fest.

Die neu erstellte Klasse ergänzen Sie wie in Listing 1 angegeben. Dort finden Sie drei private Eigenschaften namens **vorname**, **nachname** und **geburtsdatum**. Für alle drei haben wir Eigenschaftsmethoden angelegt (**Vorname**, **Nachname** und **Geburtsdatum**), mit denen die Werte für die Variablen an das Objekt übergeben und auch ausgelesen werden können.

```
using System;

namespace WPFDatenbindung {
    class Kunde {
        private string vorname;
        private string nachname;
        private DateTime geburtsdatum;
        public string Vorname {
            get { return vorname; }
            set { vorname = value; }
        }
        public string Nachname {
            get { return nachname; }
            set { nachname = value; }
        }
        public DateTime Geburtsdatum {
            get { return geburtsdatum; }
            set { geburtsdatum = value; }
        }
    }
}
```

Listing 1: Eine einfache Kunden-Klasse

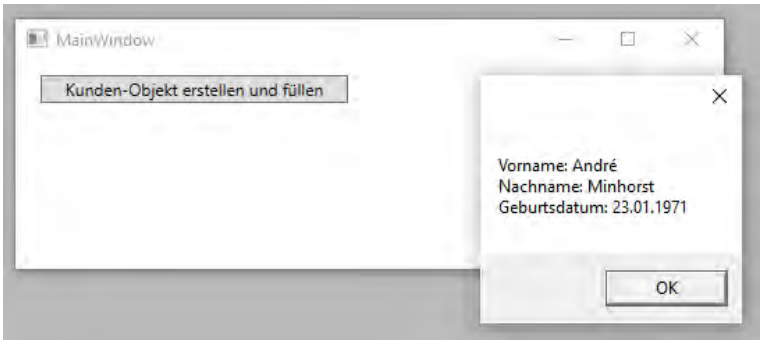


Bild 1: Erstellen eines Objekts und anzeigen seiner Werte

Kunden-Klasse testen

Um diese Klasse wie in Bild 1 zu testen, fügen wir dem Hauptfenster der Anwendung namens **MainWindow** die Schaltfläche **btnObjektErstellen** hinzu.

Für diese legen wir das Attribut **Click** mit dem Wert **btnObjektErstellen_Click** fest und erstellen gleichzeitig die gleichnamige Methode aus Listing 2 (dazu das Attribut **Click** gefolgt von Gleichheitszeichen und Anführungszeichen für das **button**-

```
private void btnObjektErstellen_Click(object sender, RoutedEventArgs e) {
    Kunde kunde = new Kunde();
    kunde.Vorname = "André";
    kunde.Nachname = "Minhorst";
    kunde.Geburtsdatum = new DateTime(1971, 1, 23);
    MessageBox.Show("Vorname: " + kunde.Vorname + "\nNachname: " + kunde.Nachname + "\nGeburtsdatum: "
        + kunde.Geburtsdatum.ToShortDateString());
}
```

Listing 2: Testen der Klasse **Kunde.cs**

Element eingeben und dann einfach mit der Tabulatortaste vervollständigen – die Methode **btnObjektErstellen_Click** finden Sie dann in der Code behind-Klasse namens **MainWindows.xaml.cs**).

Dies erstellt ein neues Objekt auf Basis der Klasse **Kunde** und füllt die drei Eigenschaften mit den entsprechenden Werten. Dabei übergeben wir das Geburtsdatum als neues Objekt des Typs **DateTime** mit dem Jahr, dem Monat und dem Tag als Parameter. Bei der Ausgabe im Meldungsfenster greifen wir über die Eigenschaft **ToShortDateString** auf das reine Datum ohne Angabe der Uhrzeit zu.

Kunden-Objekt im Fenster anzeigen

Nun wollen wir ein WPF-Fenster erstellen und zunächst die Daten eines gefüllten **Kunde**-Objekts dort abbilden. Dazu erstellen Sie ein neues WPF-Fenster namens **EinfachesObjektBinden**, wodurch auch eine Datei **EinfachesObjektBinden.xaml** plus die Code behind-Datei **EinfachesObjektBinden.xaml.cs** erzeugt werden.

Dem Fenster fügen Sie über den XAML-Code ein Grid hinzu, das aus zwei Spalten und vier Zeilen besteht (siehe Listing 3). Die drei **Label**-Steuerelemente weisen Sie über die **Grid.Column**-Eigenschaft der ersten Spalte und über **Grid.Row**

```
<Window x:Class="WPFDatenbindung.EinfachesObjektBinden" ... Title="EinfachesObjektBinden" Height="200" Width="417">
    <Grid>
        <Grid.ColumnDefinitions>
            <ColumnDefinition Width="55*"/>
            <ColumnDefinition Width="91*"/>
        </Grid.ColumnDefinitions>
        <Grid.RowDefinitions>
            <RowDefinition Height="40"/>
            <RowDefinition Height="40"/>
            <RowDefinition Height="40"/>
            <RowDefinition Height="*"/>
        </Grid.RowDefinitions>
        <Label x:Name="lblVorname" Content="Vorname:" Margin="5,5,5,5" Grid.Column="0" Grid.Row="0"/>
        <Label x:Name="lblNachname" Content="Nachname:" Margin="5,5,5,5" Grid.Column="0" Grid.Row="1"/>
        <Label x:Name="lblGeburtsdatum" Content="Geburtsdatum:" Margin="5,5,5,5" Grid.Column="0" Grid.Row="2"/>
        <TextBox x:Name="txtVorname" Grid.Column="1" Margin="5,5,5,5" Text="TextBox" />
        <TextBox x:Name="txtNachname" Grid.Column="1" Grid.Row="1" Margin="5,5,5,5" Text="TextBox" />
        <TextBox x:Name="txtGeburtsdatum" Grid.Column="1" Grid.Row="2" Margin="5,5,5,5" Text="TextBox" />
    </Grid>
</Window>
```

Listing 3: Definition des WPF-Fensters zur Anzeige der Kundendaten

den jeweiligen Zeilen zu, Gleiches gilt für die drei **TextBox**-Steuerelemente, die allerdings in der zweiten Spalte landen sollen. Um dieses Fenster zu initialisieren und anzuzeigen, fügen Sie dem Hauptfenster der Anwendung eine weitere Schaltfläche namens **btnObjektBinden** hinzu. Diese soll die folgende einfache Methode auslösen und somit das Fenster **EinfachesObjektBinden** anzeigen:

```
private void btnObjektBinden_Click(object sender,
        RoutedEventArgs e) {
    EinfachesObjektBinden wnd = new EinfachesObjektBinden();
    wnd.Show();
}
```

Steuerelemente mit Kundendaten füllen

Nun sollen die Steuerelemente die Daten eines bereits erstellten und mit Eigenschaften gefüllten Objekts auf Basis der Klasse **Kunde** anzeigen. Unter Access/VBA hätten wir dies durch die Zuweisung der entsprechenden Objekteigenschaften zu den Steuerelementen per VBA erledigt. WPF bietet jedoch für seine Fenster und Steuerelemente Eigenschaften an, mit denen Sie die Daten von Objekten per Bindung zuweisen können.

Damit die Textfelder die Daten der drei Eigenschaften Vorname, Nachname und Geburtsdatum eines **Kunde**-Objekts anzeigen, müssen wir dieses zunächst einmal initialisieren und füllen. Dies erledigen wir in der Code behind-Klasse des Fensters **EinfachesObjektBinden**, indem wir ein neues **Kunde**-Objekt mit dem **new**-Schlüsselwort erzeugen und seinen Eigenschaften direkt beim Initialisieren übergeben:

```
namespace WPFDatenbindung {
    public partial class EinfachesObjektBinden : Window {
        Kunde kunde = new Kunde {
            Vorname = "André",
            Nachname = "Minhorst",
            Geburtsdatum = new DateTime(1971, 1, 23)
        };
        public EinfachesObjektBinden() {
            InitializeComponent();
            this.DataContext = kunde;
        }
    }
}
```

Die Konstruktor-Methode **EinfachesObjektBinden**, die beim Initialisieren des Fensters automatisch ausgelöst wird, erweitern Sie wie oben zu erkennen um eine einzige Anweisung:

```
this.DataContext = kunde;
```

Damit stellen Sie die Datenherkunft für das Fenster auf das zuvor erstellte Objekt namens **kunde** ein. Die beiden Methoden werden in der abgebildeten Reihenfolge aufgerufen, die Klasse **Kunde** also bereits vor dem Zuweisen in Form des Objekts **kunde** initialisiert (davon können Sie sich beispielsweise durch das Setzen zweier Haltepunkte beim Testen der Anwendung überzeugen).

Textfelder binden

Nun fehlt noch die Anzeige der Inhalte des Objekts **kunde** in den drei Textfeldern **txtVorname**, **txtNachname** und **txtGeburtsdatum**. Dies erledigen wir, indem wir die Eigenschaft **Text** der drei Steuerelemente wie folgt füllen:

```
<TextBox x:Name="txtVorname" ...
    Text="{Binding Path=Vorname}" />
<TextBox x:Name="txtNachname" ...
    Text="{Binding Path=Nachname}" />
<TextBox x:Name="txtGeburtsdatum" ...
    Text="{Binding Path=Geburtsdatum}" />
```

Wir legen also jeweils eine Bindung zu einer der Eigenschaften des mit **DataContext** festgelegten Objekts an, wobei wir für das **Path**-Attribut den Namen der jeweiligen Objekt-Eigenschaft hinterlegen. Wenn Sie das Fenster nun testen, zeigen die drei Textfelder fast die gewünschten Daten an (siehe Bild 2). Es wäre noch praktisch, wenn auch hier das Datum im kurzen Datumsformat erschiene, also ohne Angabe der Uhrzeit. Dazu ändern Sie noch den Inhalt des **Text**-Attributs wie folgt:

WPF-Datenbindung: Listen-Objekte

Wenn Sie WPF-Anwendungen erstellen wollen, wollen Sie die Daten der Anwendung in entsprechenden Fenstern darstellen. Ein Weg zu diesem Ziel ist das Bereitstellen der Daten in Listenform, also etwa als Collection oder Dictionary. Dieser Artikel zeigt, welche Schritte nötig sind, um eine Liste von Objekten auf Basis einer einfachen Klasse zu erstellen und seine Daten in geeigneten Listen-Steuerelementen eines WPF-Fensters anzuzeigen.

Im Beitrag **WPF-Datenbindung: Einfache Objekte** haben wir uns angesehen, wie Sie ein einfaches Objekt wie einen Kunden mit Eigenschaften wie **Vorname**, **Nachname** und **Geburtsdatum** in den Textfeldern eines WPF-Fensters anzeigen. Außerdem haben Sie dort erfahren, wie Sie die Daten in beiden Richtungen synchron halten können.

Nun wollen wir ein Listenobjekt wie beispielsweise eine **Collection** mit mehreren **Kunde**-Objekten füllen und diese samt ausgewählter Eigenschaften in verschiedenen Listen-Steuerelementen von WPF anzeigen – zum Beispiel im **DataGrid**, in der **ComboBox**, in der **ListBox** oder in der **ListView**. Welche Steuerelemente für die Darstellung einer Auflistung infrage kommen, hängt übrigens allein davon ab, ob diese von der Klasse **ItemsControl** abgeleitet sind. Was die Listen-Objekte angeht, deren Inhalt wir im Listen-Steuerelement abbilden wollen, so müssen diese lediglich die **IEnumerable**-Schnittstelle implementieren.

Ob ein Steuerelement die Klasse **ItemsControl** als Basistyp verwendet, können Sie beispielsweise im Objektkatalog nachlesen. Suchen Sie dort einfach nach dem gewünschten Steuerelement und schauen sich im linken Bereich die Basistypen des Steuerelements an. In Bild 1 finden Sie beispielsweise **ItemsControl** als Basistyp für das **ComboBox**-Steuerelement vor.

Der Objektkatalog hilft auch dabei, die als Datenquelle für ein Listen-Steuerelement infrage kommenden Klassen zu identifizieren. In Bild 2 etwa erkennen wir, dass das **Collection**-Objekt die Schnittstelle **IEnumerable** implementiert und somit als Datenquelle taugt.

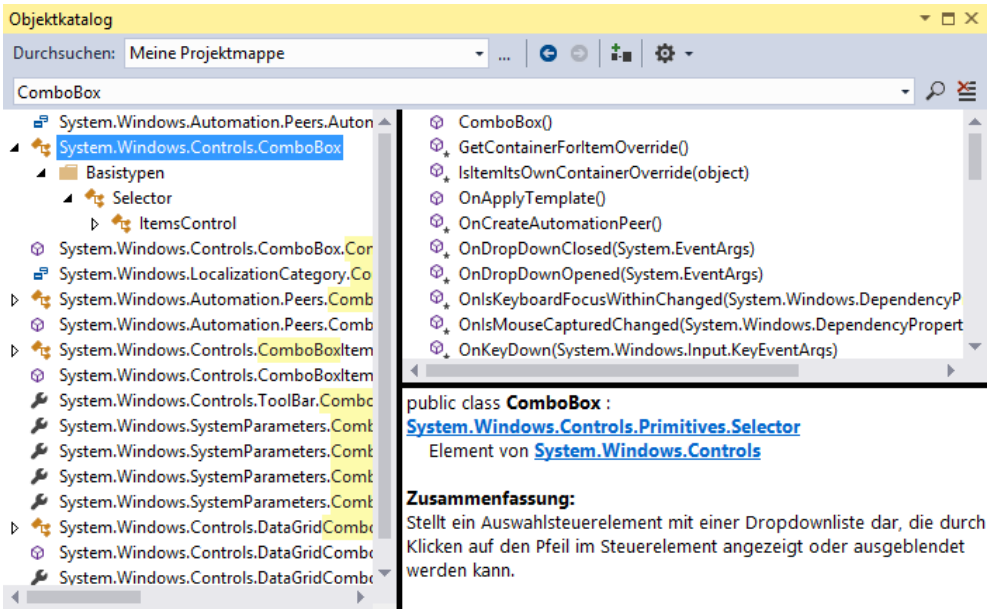


Bild 1: Prüfen, ob ein Steuerelement zur Darstellung von Listen geeignet ist

Liste mit Kunden füllen

Schauen wir uns zunächst an, wie wir ein Listen-Objekt mit einigen verschiedenen **Kunde**-Objekten füllen. Da wir in diesem Fall wissen, dass wir immer den gleichen Objekttyp zum Listen-Objekt hinzufügen möchten, verwenden wir nicht die üblicherweise verwendeten Listen-Typen wie **Collection**

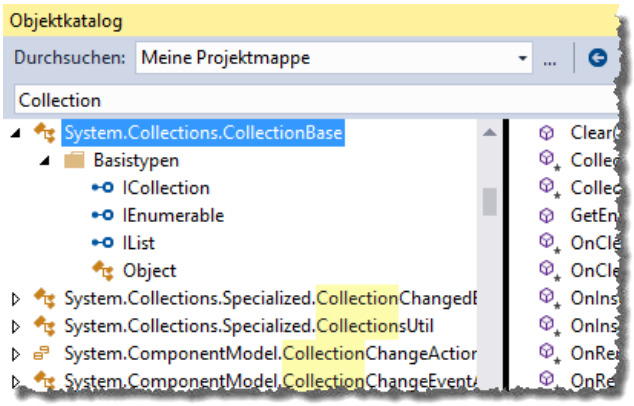


Bild 2: Untersuchen einer Klasse auf Listen-Fähigkeit

oder **ArrayList**. Diese haben nämlich den Nachteil, dass sie beliebige Objekte aufnehmen können und dadurch beim Zugriff immer eine Typkonvertierung nötig ist.

Im Gegensatz dazu gibt es die typsicheren generischen Auflistungsklassen, bei denen man direkt bei der Deklaration angibt, welchen Typ die aufzunehmenden Elemente haben. Dieser wird dabei in spitzen Klammern hinter dem Schlüsselwort für die zu verwendende Auflistungsklasse angegeben. In unserem Beispiel wollen wir eine Methode namens **GetKunden** verwenden, um eine generische Auflistungsklasse des Typs **List** mit Elementen des Typs **Kunde** zu füllen und zurückzugeben.

Um die generische Klasse **List** zu verwenden, benötigen wir die Klasse **System.Collections.Generic**, die wir mit folgender Anweisung bereitstellen:

```
using System.Collections.Generic;
```

Die Methode **GetKunden** finden Sie in Listing 1. Nach dem Deklarieren und Initialisieren des **List**-Objekts **kunden** fügt die Methode mit der **Add**-Methode nacheinander drei neue Objekte des Typs **Kunde** hinzu. Die Definition der Klasse **Kunde** entnehmen Sie dem Beispielprojekt (siehe **Kunde.cs**), eine Beschreibung dieser Klasse liefert der Artikel **WPF-Datenbindung: Einfache Objekte**.

Die Methode **GetKunden** testen wir mit einer kleinen Methode, die durch die Schaltfläche **btnListenelementeAusgeben** unseres Beispielfensters **ListeBinden.xaml** ausgelöst wird (Code siehe Listing 2). Die Methode durchläuft die **Kunde**-Objekte der mit **GetKunden** gelieferten generischen Liste in einer **foreach**-Schleife und stellt damit eine Zeichenkette zusammen, welche Vorname, Nachname und Geburtsdatum der Kunden enthält.

Bild 3: Ausgabe der Daten einer Klasse

```
private List<Kunde> GetKunden() {
    List<Kunde> kunden = new List<Kunde>();
    kunden.Add(new Kunde { Vorname = "André", Nachname = "Minhorst", Geburtsdatum = new DateTime(1971, 1, 23) });
    kunden.Add(new Kunde { Vorname = "Klaus", Nachname = "Müller", Geburtsdatum = new DateTime(1981, 2, 3) });
    kunden.Add(new Kunde { Vorname = "Barbara", Nachname = "Schmitz", Geburtsdatum = new DateTime(1976, 3, 4) });
    return kunden;
}
```

Listing 1: Füllen einer generischen Liste mit **Kunde**-Objekten

```
private void btnListenelementeAusgeben_Click(object sender, RoutedEventArgs e) {
    List<Kunde> kunden = GetKunden();
    string strKunden = "";
    foreach(Kunde kunde in kunden) {
        strKunden += "\n" + kunde.Vorname + " " + kunde.Nachname + " " + kunde.Geburtsdatum.ToShortDateString();
    }
    MessageBox.Show("Kunden:" + strKunden);
}
```

Listing 2: Testen der Methode **GetKunden**

Die Zeichenkette gibt die Methode dann per **MessageBox** auf der Benutzeroberfläche aus (siehe Bild 3).

Liste an ListBox binden

Wenn wir einmal eine Auflistung mit den gewünschten Objekten gefüllt haben, ist die Anzeige in einem Listen-Steuerelement nicht mehr besonders kompliziert. Wir wollen es zunächst mit einem **ListBox**-Steuerelement probieren.

Dazu fügen Sie zum Grid des Fensters wie in Listing 3 ein **Label**- und ein **ListBox**-Element hinzu.

Für das **ListBox**-Steuerelement legen wir vorerst keine weiteren Eigenschaften fest. Das Füllen erledigen wir mit zwei Anweisungen, die beim Öffnen des Fensters ausgeführt werden und daher in der Konstruktor-Methode der Klasse **ListeBinden.xaml.cs** landen.

Die erste Anweisung stellt die Eigenschaft **ItemsSource** auf die Methode **GetKunden** ein, die zweite erklärt das Feld **Nachname** zu dem Feld, das im Listenfeld angezeigt werden soll:

```
public ListeBinden() {
    InitializeComponent();
    lstKunden.ItemsSource = GetKunden();
    lstKunden.DisplayMemberPath = "Nachname";
}
```

Das Ergebnis sieht schon einmal den Erwartungen entsprechend aus, wie Bild 4 zeigt.

```
<Grid>
    <Grid.ColumnDefinitions>
        <ColumnDefinition Width="60" />
        <ColumnDefinition Width="*" />
    </Grid.ColumnDefinitions>
    <Grid.RowDefinitions>
        <RowDefinition Height="30" />
        <RowDefinition Height="130" />
        <RowDefinition Height="30" />
    </Grid.RowDefinitions>
    <StackPanel Orientation="Horizontal" Grid.Row="0" Grid.Column="0" Grid.ColumnSpan="2">
        <Button Content="Listenelemente ausgeben" Name="btnListenelementeAusgeben" Margin="5,5,5,5"
            Click="btnListenelementeAusgeben_Click"/>
    </StackPanel>
    <Label Content="Kunden:" Grid.Row="1" Grid.Column="0" />
    <ListBox x:Name="lstKunden" Margin="5,5,5,5" VerticalAlignment="Top" Grid.Row="1" Grid.Column="1" />
</Grid>
```

Listing 3: Definition des Grids mit dem **ListBox**-Steuerelement

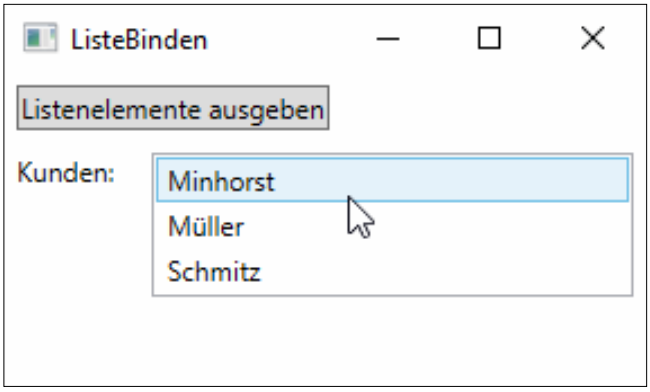


Bild 4: Anzeigen von Kunden in einer ListBox

Gewähltes Objekt anzeigen

Zunächst wollen wir uns ansehen, wie wir die Inhalte des aktuell gewählten Eintrags des **ListBox**-Steuerelements im Fenster anzeigen können. Dazu fügen wir zunächst drei Textfelder zu jeweils einer neuen Zeile im Grid hinzu.

Diese Textfelder binden wir gleich über einen speziellen Wert für das Attribut **Text** an ein Feld des aktuell in der **ListBox** ausgewählten Objekts, für das Textfeld **txtVorname** beispielsweise wie folgt:

```
<TextBox x:Name="txtVorname"
    Text="{Binding Path=Vorname}" ... />
```

Das allein hilft allerdings noch nicht, da das aktuell ausgewählte Objekt im **ListBox**-Steuerelement noch als **DataContext** angegeben werden muss. Wenn Sie nur ein einziges Textfeld an das Objekt im **ListBox**-Steuerelement binden wollten, könnten Sie das Attribut **DataContext** gleich dem entsprechenden **TextBox**-Steuerelement hinzufügen:

```
<TextBox DataContext="{Binding ElementName=lstKunden,
    Path=SelectedItem}" x:Name="txtVorname"
    Text="{Binding Path=Vorname}" ... />
```

Binding gibt an, dass eine Bindungsdefinition folgt. **ElementName** gibt das Element an, welches als Quelle dient, und **Path** die Eigenschaft, welche das Quellobjekt liefert. Hier binden wir das Attribut **Text** des **TextBox**-Steuerelements also an das Steuerelement **lstKunden**, genau genommen an die Eigenschaft **SelectedItem**. Dies liefert eine Referenz auf das aktuell markierte Element zurück, genau genommen auf ein Element des Typs **Kunde**. Auf diese Weise zeigt das Textfeld **Vorname** bereits den Vornamen des aktuell markierten Kunden im Listenfeld an.

Allerdings wollen wir ja gleich drei Textfelder mit den Werten verschiedener Eigenschaften des Objekts **Kunde** aus dem **ListBox**-Steuerelement füllen. Wir müssen aber nun nicht alle drei Textfelder mit der gleichen **DataContext**-Eigenschaft versehen, sondern können diese auch in einem übergeordneten Element festlegen – in diesem Fall etwa im **Grid**-Element:

```
<Grid DataContext="{Binding ElementName=lstKunden,
Path=SelectedItem}">
```

```
<Grid DataContext="{Binding ElementName=lstKunden, Path=SelectedItem}">
    ...
    <Label Content="Kunden:" Grid.Row="1" Grid.Column="0" />
    <ListBox x:Name="lstKunden" Margin="5,5,5,5" VerticalAlignment="Top" Grid.Row="1" Grid.Column="1" />
    <Label Content="Vorname:" Grid.Row="2" Grid.Column="0" />
    <TextBox x:Name="txtVorname" Text="{Binding Path=Vorname}" Grid.Row="2" Grid.Column="1" Margin="5,5,5,5" />
    <Label Content="Nachname:" Grid.Row="3" Grid.Column="0" />
    <TextBox x:Name="txtNachname" Text="{Binding Path=Nachname}" Grid.Row="3" Grid.Column="1" Margin="5,5,5,5"/>
    <Label Content="Geburtsdatum:" Grid.Row="4" Grid.Column="0" />
    <TextBox x:Name="txtGeburtsdatum" Text="{Binding Path=Geburtsdatum}" Grid.Row="4" Grid.Column="1" Margin="5,5,5,5"/>
</Grid>
```

Listing 4: Einstellungen für Textfelder und Grid für die Datenbindung an das aktuelle Objekt im Listenfeld

Einführung in das Entity Framework

In vorherigen Beiträgen haben wir gezeigt, wie Sie Fenster und Steuerelemente wie das DataGrid mit den Daten aus einem ADO.NET-DataSet füllen können. Wenn Sie echte mehrschichtige Anwendungen bauen möchten, gehen Sie einen Schritt weiter und nutzen einen objektrelationalen Mapper, um die Daten aus den Tabellen in Objekten abzulegen, bevor Sie diese als Datenquelle für die Benutzeroberfläche nutzen. Der Clou eines solchen Mappers ist, dass es sich um ein Framework handelt - das heißt, dass Sie nur einen geringen Teil des notwendigen Codes selbst schreiben müssen. Dieser Artikel gibt eine kleine Einführung in das Entity Framework.

Datenzugriff gestern und heute

Unter Access waren Sie es gewohnt, direkt auf die Tabellen und Abfragen der Datenbank zuzugreifen und diese beispielsweise als Datenherkunft eines Formulars oder als Datensatzherkunft von Kombinations- und Listenfeldern anzugeben. Unter VBA konnten Sie leicht mit den Methoden des **Database**-Objekts der DAO-Bibliothek wie **OpenRecordset**, **Execute** et cetera per Code auf die Daten zugreifen.

In einigen Artikeln des vorliegenden Magazins haben wir gezeigt, wie Sie per ADO.NET auf die Daten einer Datenbank zugreifen und wie Sie die dort verwendeten DataSet- und DataTable-Objekte nutzen, um die enthaltenen Daten in Fenstern und ihren Steuerelementen anzuzeigen.

Damit haben wir allerdings immer noch eine recht direkte Bindung zu den Daten der Datenbank hergestellt. Das Ziel moderner Programmiersprachen ist es jedoch, die Daten in Form von Objekten zur Verfügung zu stellen, um diese in Benutzeroberflächen etwa auf Basis von WPF (Windows Presentation Foundation) anzuzeigen und bearbeiten zu können.

Mit der Kenntnis der ADO.NET-Technologie und der Grundlagen der Objektorientierung kann man dann eine Menge Code produzieren, die dafür sorgt, dass der Inhalt etwa eines Datensatzes einer Tabelle namens **tblKunden** in ein Objekt auf Basis einer Klasse namens **clsKunde** landet. Diese stellt dann über Eigenschaften wie **Vorname**, **Nachname**, **Strasse** und so weiter alle Informationen zur Verfügung, die im entsprechenden Datensatz der Tabelle **tblKunden** gespeichert sind. Die Eigenschaften dieses Objekts können Sie nun den Steuerelementen eines Fensters zuweisen, damit der Benutzer diese ansehen und bearbeiten kann. Hat der Benutzer Änderungen an diesem Datensatz durchgeführt, sollen diese natürlich in die zugrunde liegende Datenbank übertragen werden. Dazu müssen Sie den Klassen, auf denen die Objekte basieren, entsprechende Methoden hinzufügen, die sich um das Speichern der Änderungen in die Tabellen der Datenbank kümmern.

Der benötigte Code ist kein Hexenwerk, allerdings wird es schnell recht umfangreich. Je mehr Tabellen die Datenbank enthält, desto mehr Klassen sind zu definieren, und da es meist auch noch Beziehungen zwischen den einzelnen Tabellen gibt, sind diese natürlich auch entsprechend abzubilden.

Damit Sie diesen Aufwand nicht stemmen müssen, gibt es objektrelationale Mapper wie beispielsweise das Entity Framework. Damit nimmt Ihnen Microsoft zwei wichtige Aufgaben ab: Erstens erlaubt es, auf Basis einer vorhandenen Datenbank die benötigten Klassen zu erstellen, zweitens stellt es auch noch den Code zur Verfügung, mit dem die Daten von den Tabellen in die Objekte und umgekehrt übertragen werden.

Schichtweise

Im Entity Framework spricht man von verschiedenen Schichten:

- **Konzeptionelle Schicht**: Diese Schicht entspricht den einzelnen Geschäftsobjekten der Anwendung, die in Form etwa von einzelnen Klassen/Entitäten und ihren Beziehungen untereinander erscheint. Entitäten und Beziehungen werden in der sogenannten **Conceptual Schema Definition Language** (**CSDL**) beschrieben.

- **Logische Schicht**: Die logische Schicht entspricht dem Datenmodell der Datenbank. Allerdings gibt es dafür eine eigene Sprache namens **Store Schema Definition Language** (**SSDL**). Die SSDL-Dateien enthalten die Beschreibung der Tabellen, Felder, Schlüssel, Abfragen, gespeicherten Prozeduren et cetera, allerdings in einem vom Datenbanksystem unabhängigen Format.

- **Zuordnungsschicht**: Damit fehlt noch das Bindeglied zwischen den beiden, nämlich die Zuordnungsschicht. Diese wird in der Sprache **Mapping Schema Language** (**MSL**) verfasst. Diese Schicht legt die Zuordnung zwischen der konzeptionellen Schicht und der logischen Schicht fest, und zwar wieder im XML-Format.

Von A nach B und wieder zurück

Genau genommen bietet das Entity Framework mehrere Möglichkeiten, um die Daten zu den drei Schichten zu erstellen.

Dabei kommt es darauf an, welche Voraussetzungen vorliegen:

- **Database First**: Es liegt eine Datenbank vor, auf deren Basis die drei Schichten erzeugt werden sollen. Dann besteht eine weitere Voraussetzung darin, dass es einen entsprechenden Datenprovider gibt. Dies ist beispielsweise für SQL Server-Datenbanken der Fall, für Access-Datenbanken nicht.

- **Model First**: Hier verwenden Sie den Entity Data Model-Designer von Visual Studio, um die Entitäten und ihre Beziehungen untereinander zu definieren. Anschließend erzeugen Sie auf dieser Basis die Datenbank beziehungsweise ein entsprechendes Skript.

- **Code First**: Hier erstellen Sie einfach die Klassen für die einzelnen Entitäten und legen dort die Eigenschaften und Beziehungen fest. Außerdem benötigen Sie eine weitere Klasse, mit der Sie ein paar zusätzliche Informationen angeben. Auf dieser Basis erzeugt Visual Studio dann die verschiedenen Schichten des Entity Data Models und erstellt sogar noch die Datenbank.

Für uns ist zunächst die erste Variante interessant, bei der wir das Entity Data Model auf Basis einer bestehenden Datenbank erstellen. Dazu schauen wir uns auf den folgenden Seiten ein einführendes Beispiel an.

Voraussetzungen

Voraussetzung für die hier verwendete Vorgehensweise ist Visual Studio 2015 mit Update 2. Ohne Update 2 kam es bei unseren Experimenten teilweise zu Abstürzen beim Hinzufügen einer Datenquelle auf Basis eines Objekts.

Erstellen der Anwendung

Für das Beispiel verwenden wir eine Anwendung auf Basis der Vorlage **Visual C#\WPF-Anwendung**, die Sie mit dem Dialog erstellen können, der beim Anklicken des Menüpunkts **Datei\Neu** erscheint. Unser Projekt soll **EntityBeispiel** heißen.

Entity Framework-Unterstützung hinzufügen

Das Entity Framework besteht aus einer Reihe von zusätzlichen Elementen, die Sie Ihrer Anwendung hinzufügen müssen. Um dies zu erledigen, klicken Sie mit der rechten Maustaste auf den Namen des Projekts im Projektmappen-Explorer und wählen dort den Eintrag **NuGet-Pakete verwalten...** aus. Es erscheint der **NuGet-Paket-Manager**, in dem Sie zunächst zum Bereich **Durchsuchen** wechseln. Dort sollte gleich als erster Eintrag das **EntityFramework** auftauchen, das Sie auswählen und mit einem Mausklick auf die Schaltfläche **Installieren** auf der rechten Seite installieren (siehe Bild 1).

Es erscheint ein weiterer Dialog, der den folgenden Vorgang einleitet und mit **OK** bestätigt werden kann (siehe Bild 2).

Nachdem Sie auch noch die Lizenzbedingungen bestätigt haben, installiert Visual Studio das Paket im aktuellen Projekt.

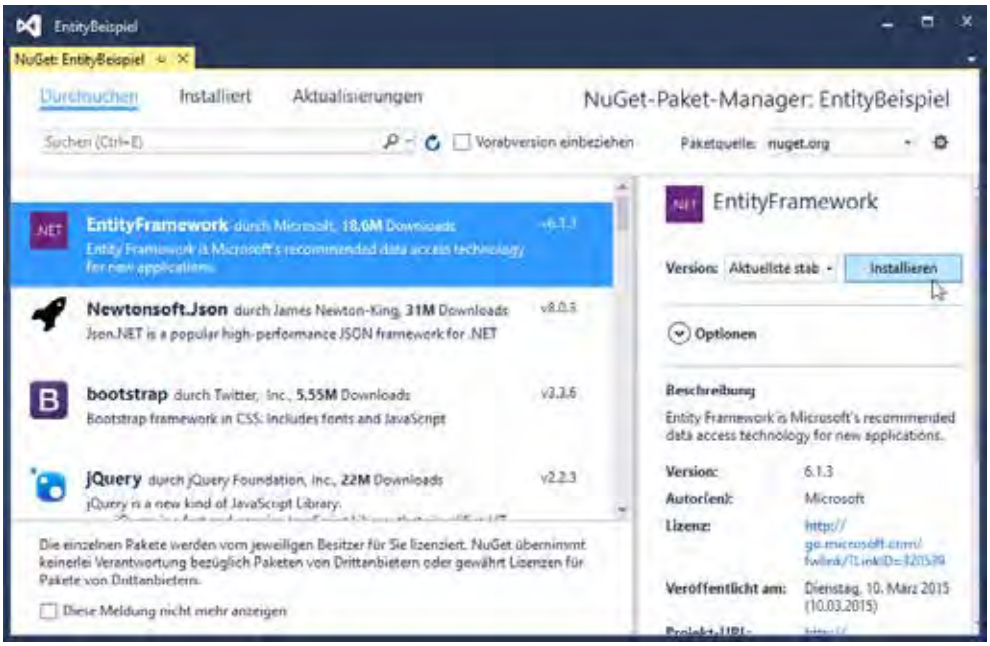


Bild 1: Hinzufügen des Entity Frameworks

Datenbank erstellen

Für den Fall, dass Sie aktuell keine SQL Server-Datenbank parat haben, die Sie als Quelle für das Beispiel nutzen können, erstellen wir schnell eine – unabhängig davon, ob Sie überhaupt SQL Server installiert haben oder nicht.

Visual Studio kommt seit der Version 2012 mit einer kleinen, für Entwickler-Zwecke geeigneten und weitgehend mit dem SQL Server kompatiblen Datenbank-Engine namens **LocalDB**. Um diese zu nutzen, aktivieren Sie mit **Ansicht|Server-Explorer** den Server-Explorer, klicken mit der rechten Maustaste auf den Eintrag **Datenverbindungen** und wählen den Kontextmenü-Eintrag **Verbindung hinzufügen...** aus (siehe Bild 3).

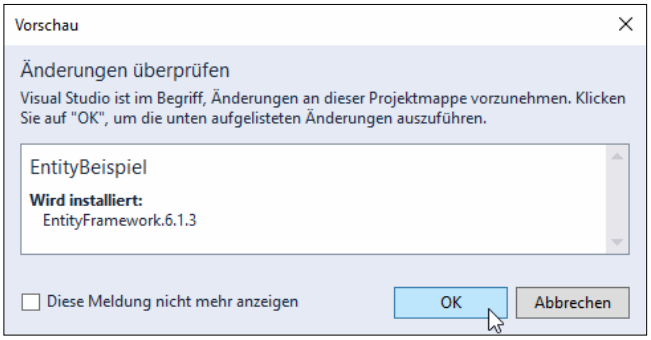


Bild 2: Installation des Entity Frameworks

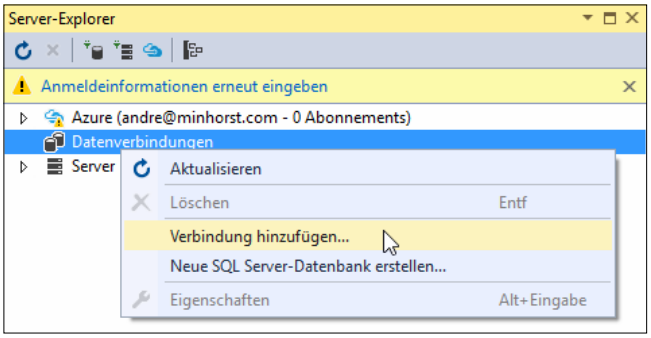


Bild 3: Hinzufügen einer neuen Verbindung

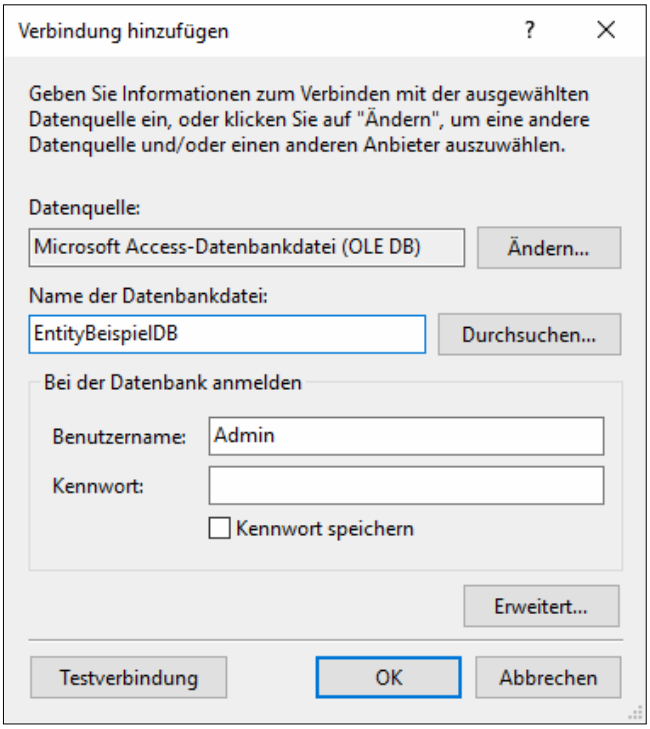


Bild 4: Hinzufügen einer neuen Datenbankverbindung

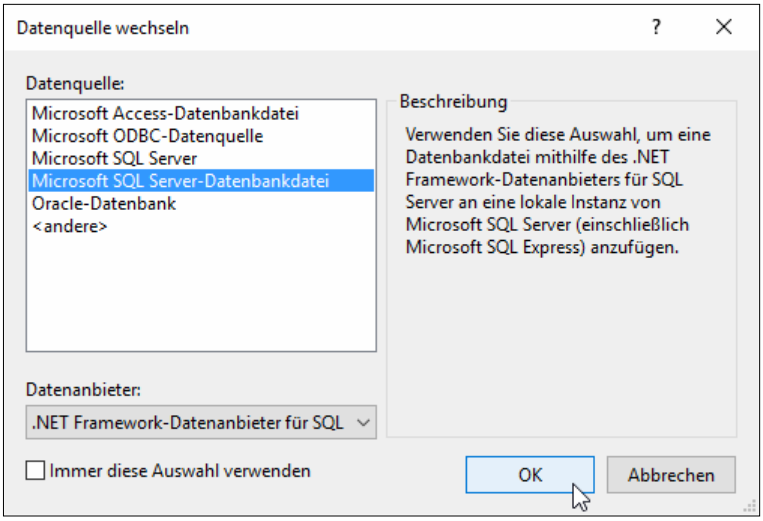


Bild 5: Auswahl des Datenquellen-Typs

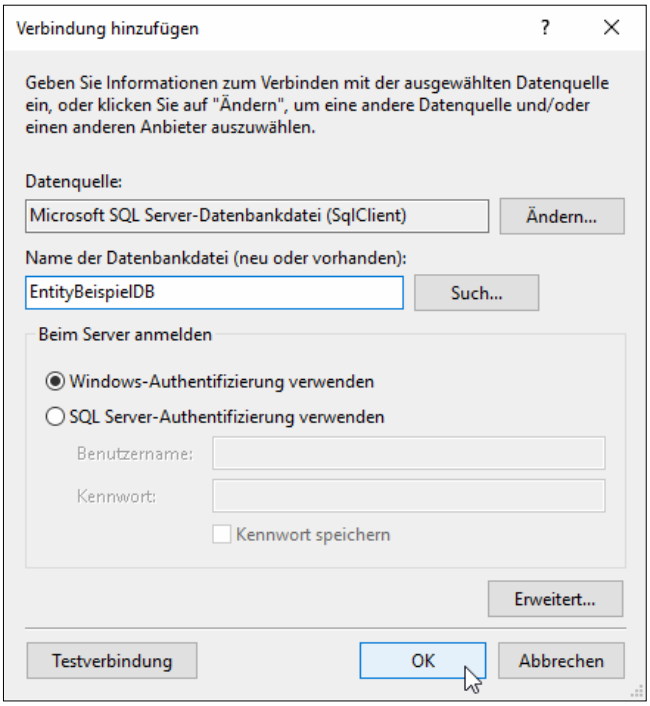


Bild 6: Angabe des Namens der neuen Datenbank

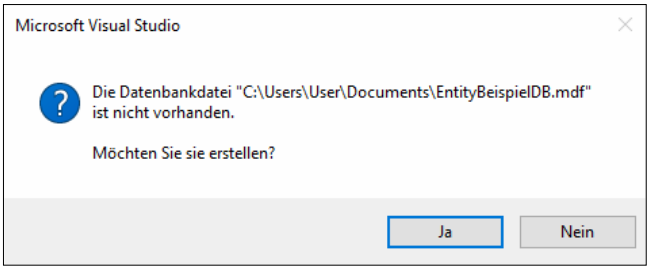


Bild 7: Visual Studio fragt, ob Sie eine neue Datenbankdatei anlegen möchten.

Im nun erscheinenden Dialog **Verbindung hinzufügen** (siehe Bild 4) klicken Sie zunächst auf die Schaltfläche **Ändern**.

Der Dialog **Datenquelle wechseln** bietet alle verfügbaren Datenquellen an, wobei Sie hier **Microsoft SQL Server Datenbankdatei** selektieren und die Auswahl mit einem Klick auf die Schaltfläche **OK** bestätigen (siehe Bild 5).

Damit kehren Sie zurück zum nun etwas anders gestalteten Dialog **Verbindung hinzufügen** (siehe Bild 6). Hier geben Sie nun den Namen der zu erstellenden Datenbankdatei ein, zum Beispiel **EntityBeispielDB**. Klicken Sie dann auf die Schaltfläche **OK**.

Es erscheint noch eine Rückfrage, ob Sie die noch nicht vorhandene Datenbankdatei anlegen möchten. Bestätigen Sie diese mit **Ja** (siehe Bild 7).

Damit erscheint die neue Datenbank nun im Server-Explorer im Bereich **Datenverbindungen** (siehe Bild 8). Wie Sie sehen, gibt es allerdings noch keine Tabellen. Aber keine Sorge – das ändern wir gleich im nächsten Schritt.

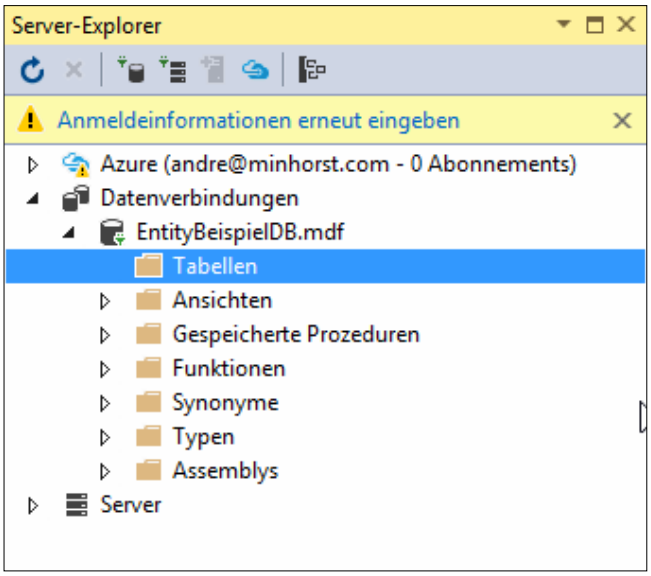


Bild 8: Die neue von **localDB** verwaltete Datenbank, hier noch ohne Tabellen

Tabellen zur Datenbank hinzufügen

Um Tabellen zur Datenbank hinzuzufügen, klicken Sie mit der rechten Maustaste auf den Eintrag **Tabellen** im Server-Explorer. Wählen Sie hier den Eintrag **Neue Abfrage** aus, um ein Fenster zur Eingabe von SQL-Code zu öffnen. Hier tragen Sie zunächst den Code aus Listing 1 ein.

```
CREATE TABLE [dbo].[tblKategorien] (
    [KategorieID] [int] NOT NULL IDENTITY,
    [Kategoriename] [nvarchar](max),
    CONSTRAINT [PK_dbo.tblKategorien] PRIMARY KEY ([KategorieID])
)
CREATE TABLE [dbo].[tblArtikel] (
    [ArtikelID] [int] NOT NULL IDENTITY,
    [Artikelname] [nvarchar](max),
    [KategorieID] [int] NOT NULL,
    CONSTRAINT [PK_dbo.tblArtikel] PRIMARY KEY ([ArtikelID]) )
CREATE INDEX [IX_KategorieID] ON [dbo].[tblArtikel]([KategorieID])
ALTER TABLE [dbo].[tblArtikel]
    ADD CONSTRAINT [FK_dbo.tblArtikel_dbo.tblKategorien_KategorieID]
    FOREIGN KEY ([KategorieID])
    REFERENCES [dbo].[tblKategorien] ([KategorieID]) ON DELETE CASCADE
```

Listing 1: Skript zum Erstellen zweier Tabellen für die Datenbank

Danach klicken Sie oben links auf das Menü mit dem Play-Symbol. Wählen Sie den Eintrag **Ausführen** aus, werden alle aktuell im Fenster angezeigten SQL-Anweisungen ausgeführt (siehe Bild 9). Alternativ können Sie sich auch die Tastenkombination **Strg + Umschalt + E** für diesen Befehl merken. Wer gelegentlich mit dem **SQL Server Management Studio** arbeitet, hat sich vielleicht an die Taste **F5** für das Starten von SQL-Code gewöhnt. Diese Taste ist in Visual Studio aber leider schon für das Starten eines Projekts zum Debuggen reserviert.

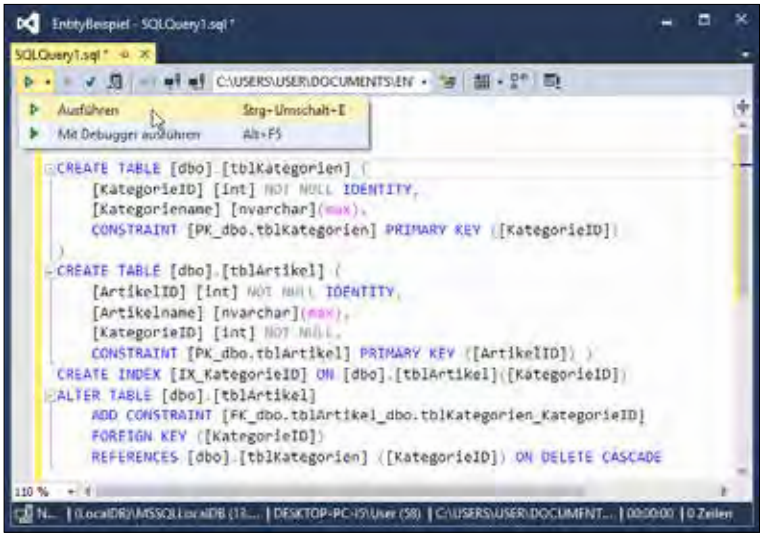


Bild 9: Ausführen des eingegebenen SQL-Codes

Die SQL-Anweisungen sollten zwei Tabellen zur Datenbank hinzufügen – eine namens **tblKategorien** mit den Feldern **KategorieID** (als Primärschlüsselfeld) und **Kategoriename** und eine namens **tblArtikel** mit den Feldern **ArtikelID** (ebenfalls als Primärschlüsselfeld), **Artikelname** und **KategorieID**. Letztere dient als Fremdschlüsselfeld zur Verknüpfung mit den Datensätzen der Tabelle **tblKategorien**.

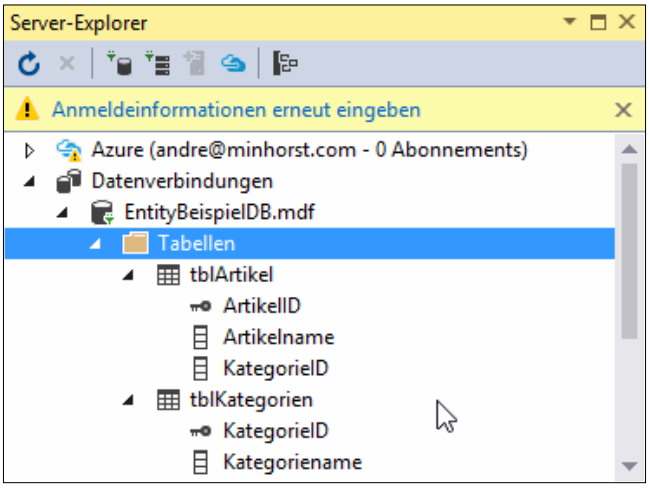


Bild 10: Der Server Explorer mit den beiden neuen Tabellen

Die erste **CREATE TABLE**-Anweisung erstellt die Tabelle **tblKategorien**, die zweite die Tabelle **tblArtikel**. Die dritte Anweisung erstellt einen Index für das Feld **KategorieID** der Tabelle **tblArtikel**, die vierte fügt die Definition des

Tipps und Tricks

Kleine Tipps und Tricks rund um die Programmierung mit Visual Studio, die keinen eigenen Artikel benötigen, landen regelmäßig in dieser Kategorie.

Kommentieren in Visual Studio/C#

Die üblichen Kommentarzeichen kennen Sie ja bereits:

- der doppelte Schrägstrich markiert den Rest der Zeile als Kommentar und
- die Kombination Schrägstrich/Sternchen und Sternchen/Schrägstrich markiert alle dazwischen liegenden Zeichen als Kommentar.

Mit dem doppelten Schrägstrich markieren Sie also eine komplette Zeile als Kommentar:

```
//Dies ist eine Kommentarzeile.
```

Oder nur der hintere Teil soll als Kommentar interpretiert werden:

```
Console.WriteLine(); //Leere Zeile
```

Mehrere Zeilen schließen Sie entsprechend ein:

```
/*Dies ist ein
mehrzeiliger Kommentar.*/
```

Sie können allerdings auch einen Kommentar mitten in einer Codezeile unterbringen:

```
int /*Mittendrin*/ Test;
```

Ob und wo dies sinnvoll ist, entscheiden Sie selbst. Vielleicht kann man so Stellen im Code markieren, an denen noch etwas geändert werden muss.

Tastenkombinationen für Kommentare

Wenn Sie eine komplette Zeile oder mehrere Zeilen schnell als Kommentar markieren möchten, betätigen Sie einfach die Tastenkombination **(Strg + K) (Strg + C)**. Um als Kommentarzeilen markierte Zeilen wieder auszukommentieren, verwenden Sie die Tastenkombination **(Strg + K) (Strg + U)**. Zum Auskommentieren einer kompletten Zeile reicht es, wenn sich vor dem Betätigen der Tastenkombination die Einfügemarke in der Zeile befindet. Für mehrere Zeilen muss sich die Markierung über die betroffenen Zeilen erstrecken, wobei die erste und letzte Zeile nicht komplett markiert sein müssen.

Wenn Sie nur ein paar Zeichen innerhalb einer Zeile markieren möchten, also beispielsweise einige Zeichen mitten in der Zeile oder die hinteren Zeichen der Zeile, markieren Sie den Bereich und betätigen dann wiederum **(Strg + K) (Strg + C)**. Die Zeichen werden dann in **/*** und ***/** eingefasst – auch wenn sich die Markierung am Ende der Zeile befindet und theoretisch das Einleiten des auskommentierten Bereichs mit **//** ausgereicht hätte.

Die Auskommentierung mehrerer Zeilen mit **/*** und ***/** erreichen Sie übrigens nur, wenn Sie dies manuell eingeben – das Markieren mehrerer Zeilen und das Betätigen der Tastenkombination **(Strg + K) (Strg + C)** versieht mehrere markierte Zeilen immer mit führendem **//**.

Wer Tastenkombinationen nicht so gern nutzt, kann den markierten Bereich auch mit den beiden Schaltflächen der Symbolleiste Text-Editor ein- und auskommentieren (siehe Bild 1).

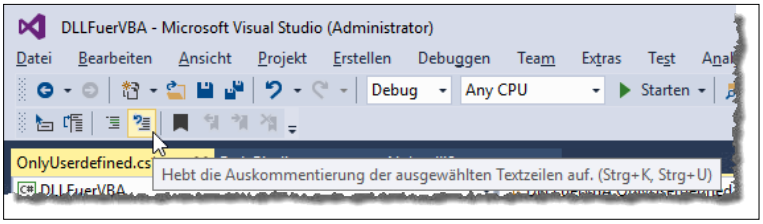


Bild 1: Ein- und Auskommentieren per Menüleiste

Einer Anwendung beim Start Parameter übergeben

Wenn Sie eine **.exe**-Datei erstellen, beispielsweise als Konsolenanwendung, möchten Sie dem Benutzer vielleicht die Übergabe von Parametern ermöglichen. Dazu ergänzen Sie die Methode **Main** um den Code aus Listing 1.

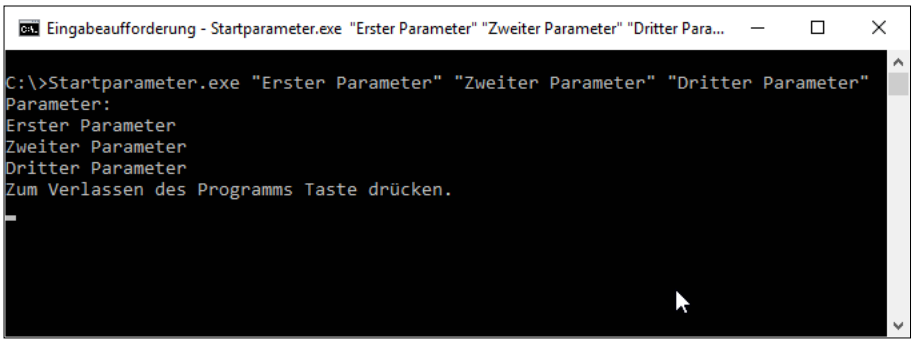


Bild 2: Ausprobieren der Parameterübergabe

Die Methode erwartet mit dem Parameter **args** eine Liste von **String**-Argumenten. Die **if**-Bedingung prüft mit dem Vergleich der Eigenschaft **args.Length** mit dem Wert **0**, ob **args** mehr als **0** Elemente enthält. Ist dies der Fall, durchläuft eine **for**-Schleife die enthaltenen Elemente. Dazu verwendet sie als Startwert für die Laufvariable **i** den Wert **0** und erhöht diesen mit jedem Durchlauf um den Wert **1**

Solange **i** kleiner als die Anzahl der enthaltenen Elemente ist, gibt die Methode den Inhalt des Arrays **args** für den jeweiligen Indexwert aus der Variablen **i** in der Konsole aus. Bei drei Elementen durchläuft die Schleife also die Werte **0**, **1** und **2**. Der Wert **3** ist nicht mehr kleiner als die Anzahl der Elemente, also wird die Schleife nicht erneut durchlaufen.

Um dies selbst auszuprobieren, legen Sie ein neues Projekt auf Basis der Vorlage **Visual C#|Konsolenanwendung** an und ergänzen den Code der Methode **Main** wie hier abgedruckt. Betätigen Sie dann den Menübefehl **Erstellen|Startparameter erstellen** (vorausgesetzt Sie haben das Projekt ebenfalls **Startparameter** genannt).

Sie können die dabei erzeugte **.exe**-Datei **Startparameter.exe** dann aus dem Projektverzeichnis **C:\Users\<Benutzername>\...\Startparameter\bin\Debug** etwa nach **c:** kopieren und dann mit folgendem Befehl aufrufen:

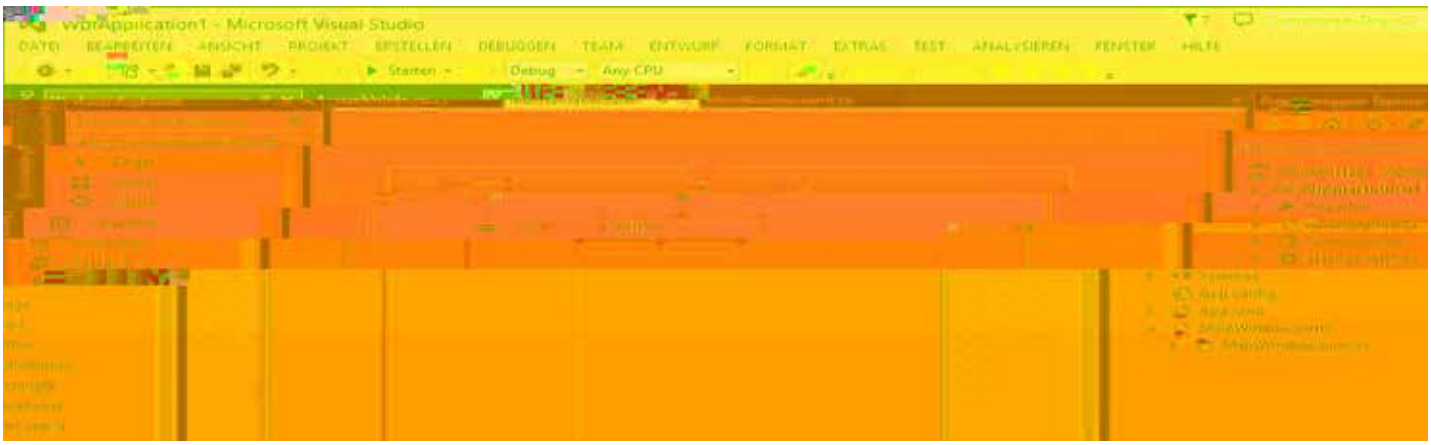
```
Startparameter.exe "Erster Parameter" "Zweiter Parameter"
"Dritter Parameter"
```

```
class Program {
    static void Main(string[] args) {
        if (args.Length > 0 ) {
```

AUSGABE 04/2016

DATENBANK ENTWICKLER

MAGAZIN FÜR DIE DATENBANKENTWICKLUNG MIT VISUAL STUDIO FÜR DESKTOP, WEB UND CO.



TOP-THEMEN:

SQL SERVER	Datenbanken erstellen mit dem SSMA	SEITE 8
SQL SERVER	SQL Server-Interaktion mit dem Profiler verfolgen	SEITE 21
DATENZUGRIFF	Entity Data Model für eine Datenbank erstellen	SEITE 26
DATENZUGRIFF	LINQ to Entities: Daten abfragen	SEITE 35
VON VBA ZU WPF	WPF/EDM: Kundenübersicht	SEITE 51

LocalDB-Datenbanken nutzen

LocalDB ist ein spezieller Ausführungsmodus von SQL Server Express, den Sie gut für die Entwicklung und die Weitergabe von Datenbankanwendungen auf Basis von .NET-Technologien nutzen können. Außerdem ist diese Variante des SQL Servers wesentlich leichtgewichtiger und in weniger als einer Minute zu installieren. Der Download umfasst gerade mal 45 MB.

LocalDB herunterladen

Den direkten Download, der nur die Installationsdatei für **LocalDB** enthält, finden Sie beispielsweise unter folgendem Link:

```
https://download.microsoft.com/download/E/1/2/E12B3655-D817-49BA-B934-CEB9DAC0BAF3/SqlLocalDB.msi
```

LocalDB installieren

Nachdem Sie die nur rund 45 MB große Datei **SqlLocalDB.msi** heruntergeladen haben, können Sie diese gleich durch einen Doppelklick installieren. Dazu brauchen Sie nur wenige Schritte im Installationsassistenten zu absolvieren (siehe Bild 1).

Alternativ führen Sie die Installation ohne weitere Interaktion über die Eingabeaufforderung oder eine entsprechende Batch-Datei aus.

Die Eingabeaufforderung muss dabei mit Administratorrechten geöffnet werden. Dazu geben Sie zunächst **cmd** in

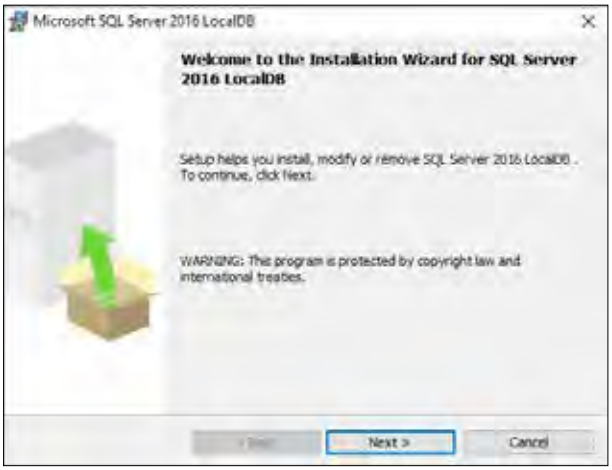


Bild 1: Start der Installation

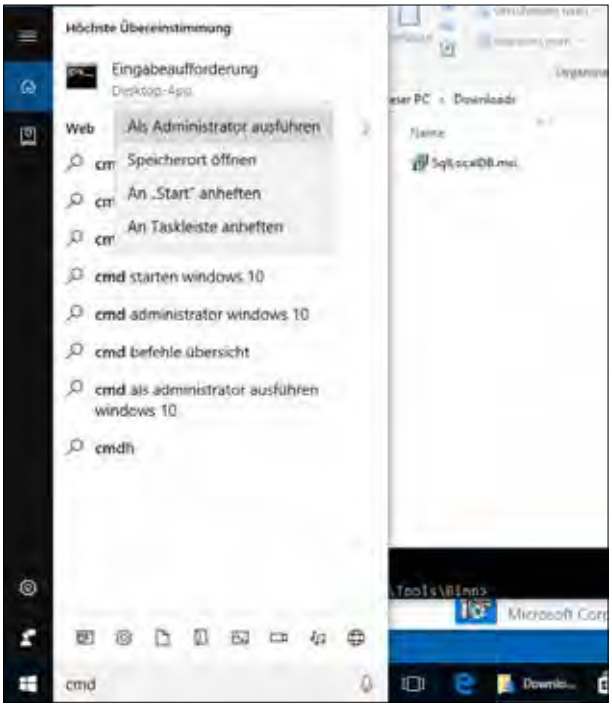


Bild 2: Start der Eingabeaufforderung als Administrator

das Feld **Web und Windows durchsuchen** von Windows ein. Dort erscheint dann ein Eintrag namens Eingabeaufforderung, auf den Sie mit der rechten Maustaste klicken. Im Kontextmenü wählen Sie dann den Eintrag **Als Administrator ausführen** aus (siehe Bild 2).

Hier geben Sie nun die folgende Anweisung ein, um LocalDB ohne weitere Rückfragen zu installieren:

```
msiexec /i SqlLocalDB.msi /qn IACCEPTSQLLOCALDBLICENSETERMS=YES
```

Ob **LocalDB** installiert wurde, können Sie nun beispielsweise in der Systemsteuerung unter **Programme und Features** prüfen (siehe Bild 3).

LocalDB testen

Was machen wir nun mit LocalDB? Als Erstes könnten wir herausfinden, wie die Instanz benannt ist, damit wir Verbindungszeichenketten zusammensetzen oder per **SQL**

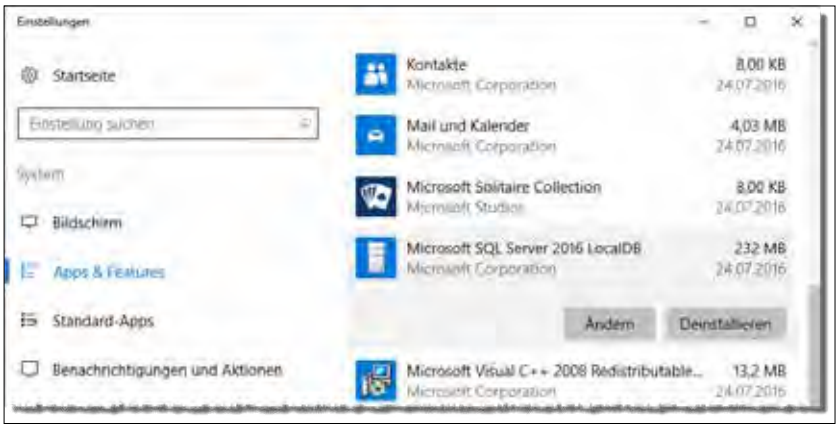


Bild 3: LocalDB wurde erfolgreich installiert.

Server Management Studio darauf zugreifen können. Dazu bleiben Sie gleich in der Eingabeaufforderung und setzen beispielsweise den folgenden Befehl ab:

```
sqllocaldb info
```

Dies sollte den Namen der Standardinstanz zurückgeben:

```
MSSQLLocalDB
```

Wenn Sie LocalDB wie oben beschrieben allein installiert haben und Sie diese nicht etwa mit Visual Studio oder einer SQL Server Installation auf den Rechner gebracht haben, erscheint vielleicht auch diese Meldung:

```
Der Befehl "sqllocaldb" ist entweder falsch geschrieben
oder konnte nicht gefunden werden.
```

In diesem Fall wechseln Sie zum Verzeichnis, in dem sich die Datei **sqllocaldb.exe** befindet, normalerweise in diesem hier:

```
C:\Program Files\Microsoft SQL Server\130\Tools\Binn
```

Wenn Sie nicht jedesmal in dieses Verzeichnis wechseln wollen, können Sie das Verzeichnis auch in die Umgebungsvariable **Path** aufnehmen. Dadurch fügt Windows das Verzeichnis zum Suchpfad hinzu, der immer bei Eingabe von Befehlen in die Kommandozeile durchsucht wird.

Geben Sie im Feld **Web und Windows durchsuchen** den Ausdruck **Path** ein und klicken Sie dann oben auf **Systemumgebungsvariablen bearbeiten**. Klicken Sie im nun erscheinenden Dialog **Systemeigenschaften** auf die Schaltfläche **Umgebungsvariablen**.

Es erscheint ein weiterer Dialog namens **Umgebungsvariablen**. Je nachdem, ob Sie den Pfad zur Datei **sqllocaldb.exe** nur für den aktuell angemeldeten Benutzer oder systemweit eintragen möchten, markieren Sie im oberen oder im unteren Bereich den Eintrag **Path**.

Klicken Sie dann auf **Hinzufügen** und kopieren Sie den Pfad zur Datei **sqllocaldb.exe** dort hinein. Nach dieser Aktion müssen Sie sich gegebenenfalls abmelden und erneut anmelden oder das System neu starten. Wenn Sie nun **sqllocaldb info** in die Eingabeaufforderung eingeben, erhalten Sie das Ergebnis aus Bild 4. Die Ausgabe entspricht dem Namen der Standardinstanz von **localdb**.

Damit wissen Sie auch gleich, wie Sie dieses Programm als Datenbankserver etwa in Verbindungszeichenfolgen oder auch im SQL Server Management Studio ansprechen können – nämlich mit **(localdb)\MSSQLLocalDB**.

Wenn Sie eine Übersicht aller Befehle von **sqllocaldb** wünschen, geben Sie einfach **sqllocaldb** in die Eingabeaufforderung ein.

Sollten Sie eine neue Instanz von **localdb** erzeugen wollen, erledigen Sie dies mit dem **create**-Parameter und geben den Namen der benannten Instanz an:

```
C:\Users\andre>sqllocaldb create MeineInstanz
Die LocalDB-Instanz "MeineInstanz" wurde mit Version
13.0.1601.5 erstellt.
```

Die neue Instanz wird dann mit dem Parameter **info** ausgegeben:

Datenbanken erstellen mit dem SSMA

Wer von Access kommt und nun Anwendungen auf Basis von WPF, C# und dem Entity Framework erstellen möchte, wird feststellen, dass dies mit Access-Datenbanken als Backend nicht gelingt – es lässt sich schlicht kein Entity Framework für Access-Datenbanken erstellen. Da kann es dann hilfreich sein, wenn Sie, nachdem wir uns bereits mit der Migration von Access zum SQL Server beschäftigt haben, auch neue Datenbanken im SQL Server erstellen können. Dieser Artikel zeigt die wichtigsten Techniken für die Erstellung von Datenbanken über die Benutzeroberfläche des Microsoft SQL Server Management Studios (SSMA).

Mit der für kleine Unternehmen kostenfreien Community Edition des SQL Servers oder auch mit den kostenpflichtigen Versionen kommt auch das SQL Server Management Studio auf Ihren Rechner. Das ist ein Tool, mit dem Sie Datenbanken verwalten können – und dazu gehört natürlich auch das Erstellen neuer Datenbanken.

Datenbanken für das Entity Framework

Wenn Sie einmal eine Migration einer reinen Access-Datenbank zu einer Kombination aus Access-Frontend und SQL Server-Backend durchgeführt haben, waren Sie sicher nicht mit der Migration der Tabellen zufrieden. Stattdessen haben Sie auch noch Stored Procedures, Trigger und weitere SQL Server-spezifische Objekte erstellt, um Geschäftslogik auf den SQL Server zu transferieren.

Bei Verwendung des Entity Frameworks ist das nicht notwendig. Hier reicht es aus, wenn die Tabellen mit den Daten auf dem SQL Server landen, die Geschäftslogik soll in anderen Schichten der Anwendung untergebracht werden. Wie und wo dies geschieht, werden wir uns in späteren Artikeln ansehen.

Fürs Erste reicht es uns, zu wissen, dass wir vom Entity Framework aus direkt auf die Tabellen der Datenbank zugreifen und dass diese die Informationen liefern, um das Entity Framework auf Basis der Tabellen und ihrer Beziehungen untereinander aufbauen zu können.

Dies ist erfordlich, wenn Sie das Entity Framework auf Basis der Tabellenstruktur erstellen wollen. Der Assistent kann dann Informationen über Primärschlüssel- und Fremdschlüsselfelder direkt in entsprechende Beziehungen umwandeln.

Benennung von Tabellen und Feldern

Der Entity Framework-Assistent übernimmt die Bezeichnungen von Tabellen und Feldern für die zu erstellenden Klassen. Das heißt, dass Sie sich wohl oder übel von einigen unter Access liebgewonnenen Gewohnheiten verabschieden sollten.

Dort haben Sie Tabellen vermutlich mit dem Präfix **tbl** ausgestattet und den Rest der Tabelle entsprechend dem Plural der enthaltenen Elemente benannt, also beispielsweise **tblKunden**.

Dies würde dazu führen, dass die Klasse im Entity Framework, welche die Datensätze der Tabelle **tblKunden** abbildet, automatisch auch **tblKunden** heißen würde. Das entspricht nicht den gängigen Benennungen für Klassen, die erstens im Singular und zweitens ohne Präfix erscheinen sollten – in diesem Fall also einfach **Kunde**.

Nun gibt es im Entity Framework-Assistenten sogar eine Option, mit der man abfangen kann, dass die Tabellen entsprechend dem Plural der gespeicherten Elemente benannt sind.

Dabei ist die Option allerdings nur auf den Plural für englische Benennungen optimiert (und somit auf das angehängte **s** für den Plural), sodass **Customers** in **Customer** geändert werden kann. Bei **Kunden** und **Kunde** sieht es anders aus, sodass wir zwei Möglichkeiten für unsere deutschsprachig

benannten Tabellen haben: entweder wir benennen diese direkt im Singular oder wir vergeben Plural-Namen. In beiden Fällen müssen später etwa bei Verwendung im Entity Framework noch die eine oder andere individuelle Anpassung vornehmen.

Wenn Sie eine Anwendung auf einer bereits vorhandenen Datenbank aufbauen müssen, bleibt Ihnen aber ohnehin keine Wahl – dann sind spätere Optimierungen sowieso unumgänglich, wenn Sie mit vernünftigen Klassennamen arbeiten wollen.

In diesem Artikel wollen wir uns auf eine Datenbank konzentrieren, die auf der neuen **LocalDB**-Instanz läuft – ein Feature, das seit der Version 2012 mit Visual Studio geliefert wird.

Um mit dem SQL Server Management Studio zu arbeiten, benötigen Sie eine entsprechende Installation. Die Software können Sie auf verschiedene Arten auf den Rechner bringen, beispielsweise durch Installation eines entsprechenden SQL Server-Pakets (siehe Artikel **SQL Server 2014 Express installieren**).

Wenn Sie, wie im vorliegenden Artikel beschrieben, eine neue SQL Server-Datenbank mit der Datenbank-Engine **LocalDB** erstellen wollen, brauchen Sie nicht das komplette SQL Server-Paket, das Ihnen nicht nur eine Menge Speicherplatz auf der Festplatte belegt, sondern auch eine Menge Zeit bei der Installation raubt. Also installieren Sie einfach nur das SQL Server Management Studio, das Sie unter dem folgenden Link finden:

```
https://msdn.microsoft.com/de-de/library/
mt238290.aspx
```

Dies liefert die Datei **SSMS-Setup-DEU.exe**, welche Sie einfach ausführen.

Mit LocalDB verbinden

Um sich mit dem als Datenbankserver verwendeten LocalDB zu verbinden (oder auch mit einem anderen Datenbankserver wie etwa der Vollversion des SQL Servers), starten Sie das

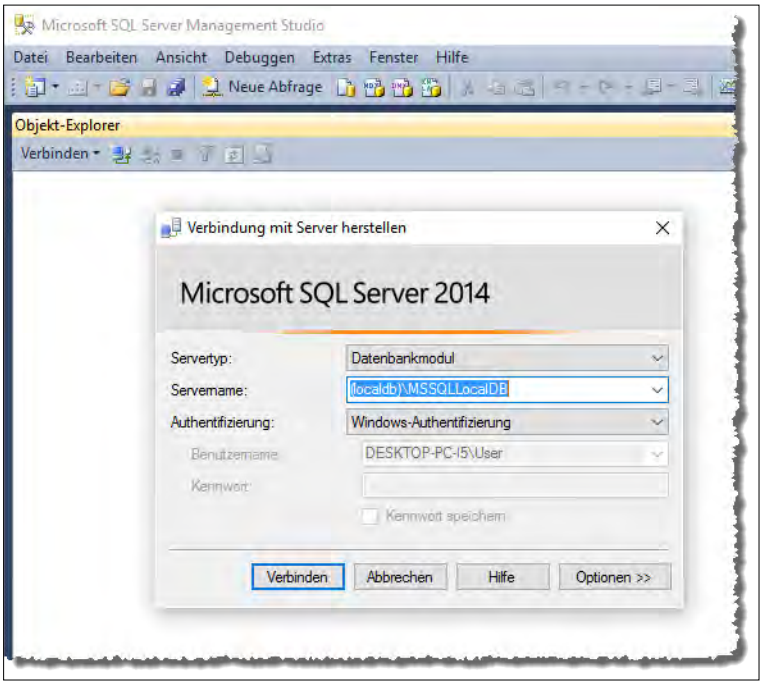


Bild 1: Mit dem SQL Server verbinden – hier mit **LocalDB**

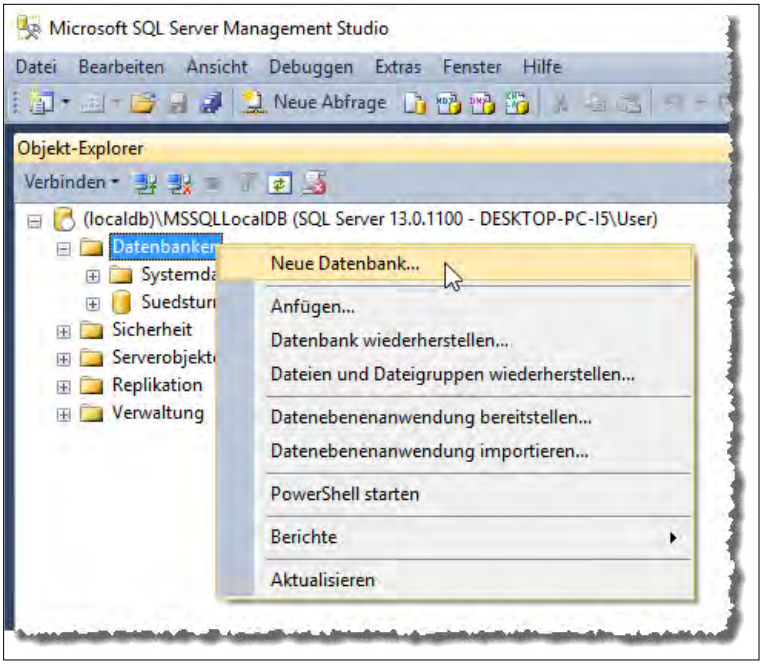


Bild 2: Anlegen einer neuen Datenbank

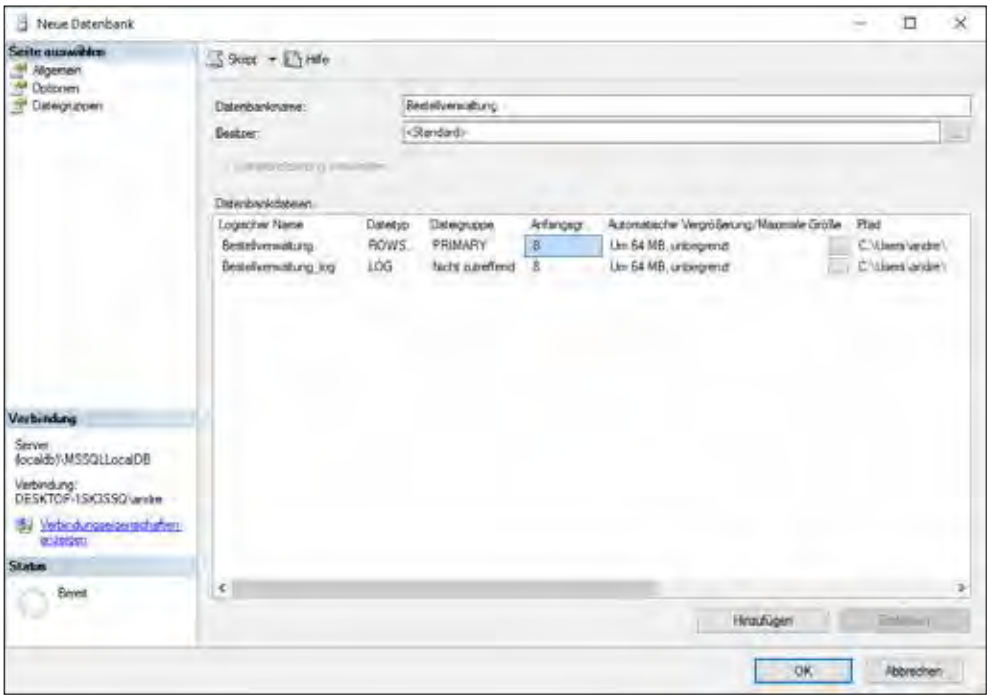


Bild 3: Angabe des Datenbanknamens

SQL Server Management Studio. Es erscheint der Dialog aus Bild 1, der Sie zur Eingabe des Servernamens auffordert.

Verwenden Sie die Standardinstanz von **LocalDB**, geben Sie einfach folgenden Ausdruck ein:

```
(localdb)\MSSQLLocalDB
```

Sie können auch das Kombinationsfeld aufklappen, um die Suche nach den verfügbaren SQL Servern zu starten, aber dies nimmt eine Menge Zeit in Anspruch – schneller geht es mit der direkten Eingabe.

Für den Zugriff auf eine benannte Instanz von LocalDB geben Sie statt **MSSQLLocalDB** den entsprechenden Namen der Instanz an.

Datenbank anlegen

Nach erfolgreichem Verbindungsaufbau können Sie gleich eine neue Datenbank anlegen, und zwar über den Eintrag **Neue Datenbank** des Kontextmenüs des Elements **Datenbanken** im Objekt-Explorer (siehe Bild 2).

Der folgende Dialog namens **Neue Datenbank** erwartet die Eingabe der Bezeichnung für die Datenbank (siehe Bild 3). Geben Sie hier den Text **Bestellverwaltung** an und klicken Sie auf **OK**, um die Änderung zu übernehmen und den Dialog zu schließen. Unter **Pfad** können Sie außerdem angegeben, in welchem Verzeichnis die beiden Datenbankdateien gespeichert werden sollen.

Tabellen anlegen

Die erste Tabelle legen Sie nun an, indem Sie unter **Bestellverwaltung** mit der rechten Maustaste auf das Element **Tabellen** klicken und den Eintrag **Tabelle...** des Kontextmenüs auswählen (siehe Bild 4).

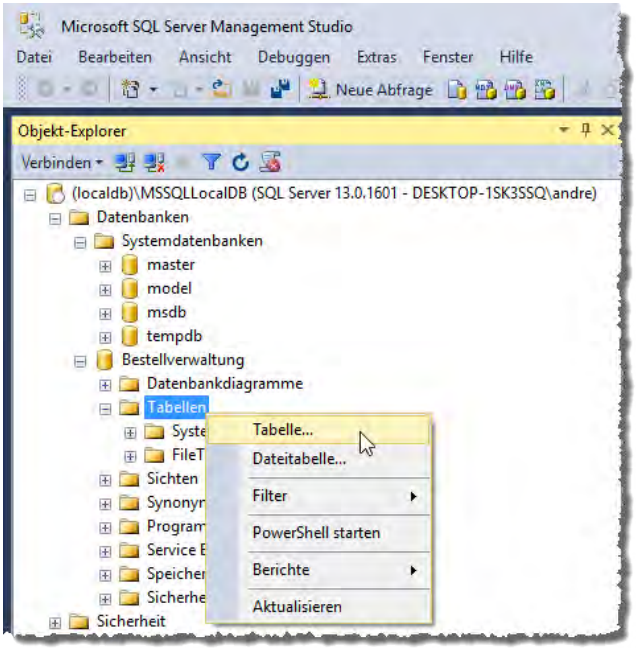


Bild 4: Neue Tabelle anlegen

Es erscheint nun die Entwurfsansicht der neuen Tabelle.

Diese liefert drei wichtige Bereiche (siehe Bild 5):

- Links oben finden Sie die Liste der Felder mit den Feldnamen, Felddatentypen und der Angabe, ob für das Feld Nullwerte zulässig sind.
- Links unten zeigt der SSMA die Eigenschaften des aktuell markierten Feldes beziehungsweise der Spalte an. Hier finden sich noch weitere Eigenschaften zu den einzelnen Feldern.
- Rechts sehen Sie die Eigenschaften der aktuell bearbeiteten Tabelle. Hier tragen Sie beispielsweise den Tabellennamen, eine Beschreibung et cetera ein.

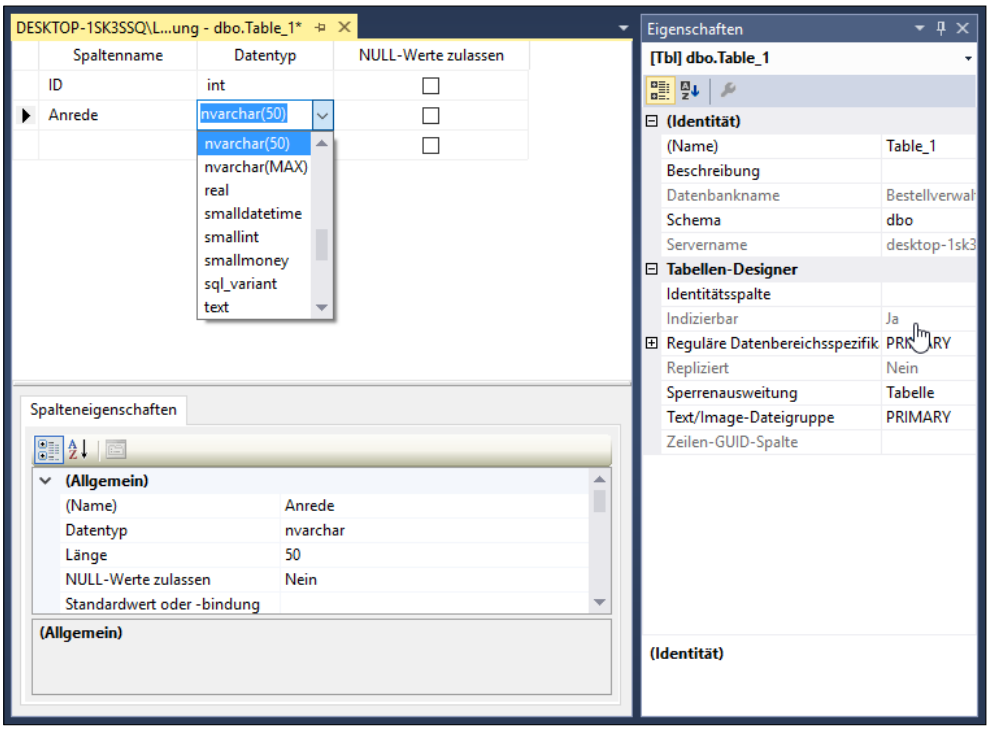


Bild 5: Entwurfsansicht einer neuen Tabelle

Erste Tabelle: Anreden

Die erste Tabelle soll gemäß der weiter oben besprochenen Konvention nach dem enthaltenen Element im Plural benannt werden, im Falle einer Tabelle zum Speichern von Anreden also schlicht **Anreden**. Tragen Sie diesen Namen für die Eigenschaft **Name** der Tabelle im rechten Bereich ein. Der Artikel **Datenzugriff per Entity Framework** zeigt, wie wir die Tabellen der hier erstellten Datenbank per Entity Framework nutzen und welche Änderungsschritte notwendig sind, wenn Sie die Tabelle anders als **Anreden** nennen, aber im Entity Framework ein Objekt namens **Anrede** verwenden wollen.

Primärschlüsselfeld anlegen

Als erstes Feld legen wir ein Primärschlüsselfeld an. Auch hier sollten Sie, genau wie beim Tabellennamen, eine Konvention befolgen. Die beiden gängigsten Varianten lauten, für das Primärschlüsselfeld einer Tabelle immer die Bezeichnung **ID** zu verwenden oder noch den Namen des enthaltenen Elements voranzustellen, also **AnredeID**. Wie auch immer Sie sich entscheiden, sollten Sie sicherstellen, dass Sie bei Ihrer selbst gewählten Benennungskonvention bleiben.

Wenn man sich jedoch anschaut, wie die verschiedenen Varianten später beim objektorientierten Zugriff auf die jeweiligen Objekte und Eigenschaften aussehen, liegt die Wahl der kürzeren Variante, nämlich ID für das Primärschlüsselfeld, nahe. Dann würde der Zugriff etwa über **Anrede.ID** erfolgen und nicht über **Anrede.AnredeID**, was ja doch irgendwie doppelt gemoppelt ist.

Das Primärschlüsselfeld soll keine Null-Werte zulassen, also deaktivieren Sie das Kontrollkästchen für die Zeile **AnredeID** und die Spalte **NULL-Werte zulassen**.

Den Datentyp des Primärschlüsselfeldes stellen Sie auf **int** ein. Danach wollen wir noch die Autowert-Eigenschaft

umsetzen. Dazu wechseln Sie zum Bereich links unten, wo sich die Spalteneigenschaften befinden. Dort finden Sie die Eigenschaft **Identitätsspezifikation**, für die Sie weitere Untereigenschaften aufklappen können. Dort legen Sie **(Ist Identity)** auf **Ja** fest und behalten die Werte für **ID-Ausgangswert** und **ID-Inkrement** bei (siehe Bild 6).

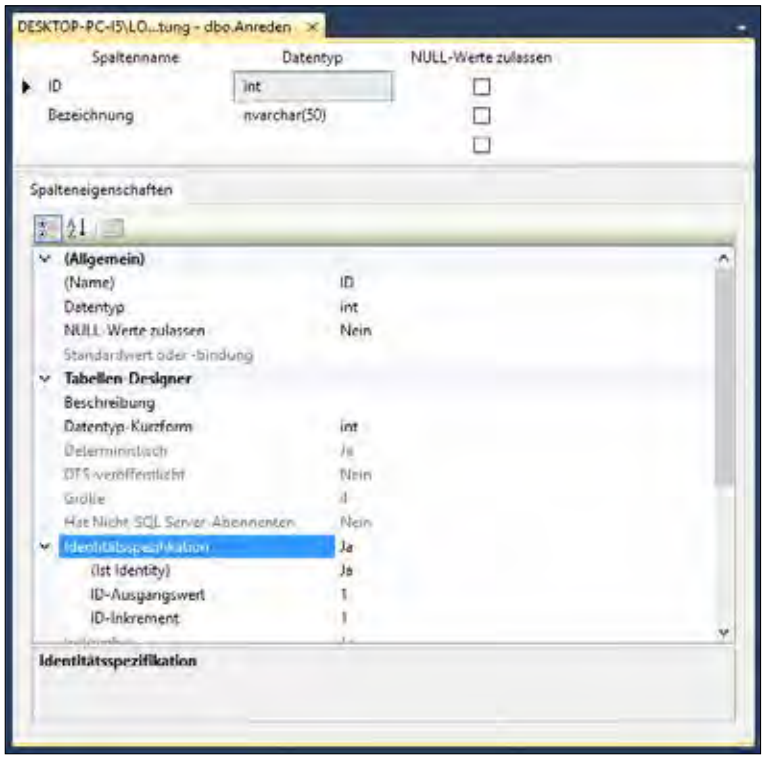


Bild 6: Festlegen der Identitätsspezifikation

Damit ist das Feld **AnredeID** theoretisch aber immer noch kein Primärschlüsselfeld. Dies stellen Sie schließlich über den Kontextmenüeintrag **Primärschlüssel festlegen** ein (siehe Bild 7).

Die als Primärschlüsselfeld definierte Spalte sieht schließlich wie in Bild 8 aus.

Weiteres Feld definieren

Das zweite Feld dieser Tabelle soll die Anrede aufnehmen. Hierfür legen wir den Datentyp **nvarchar(50)** fest. Auch dieses Feld soll keine Null-Werte enthalten. Außerdem definieren wir für diese Spalte die Bezeichnung **Anrede** – welche auch sonst? Nun, es gäbe noch die Möglichkeit, etwa auf einen Wert wie **Name** oder **Bezeichnung** auszuweichen. Unter Access gab es aber immer Probleme mit Feldnamen, die wie Schlüsselwörter verschiedener Funktionen hießen – und **Name** ist nun einmal die Bezeichnung einiger Eigenschaften im Access-Bereich.

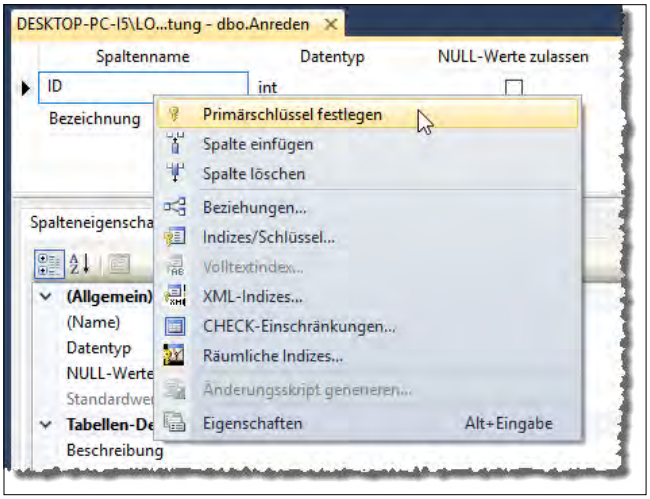


Bild 7: Primärschlüssel der Tabelle festlegen

Sind bei Verwendung etwa als Datenquelle für das Entity Framework ähnliche Probleme zu erwarten? Ja – es gibt eine Einschränkung bei der Vergabe von Feldnamen, zumindest wenn diese später als Eigenschaften im Entity Framework genutzt werden sollen: Ein Feldname darf dort nicht mit dem Namen der Klasse beziehungsweise der Entität übereinstimmen, sonst wird dieser automatisch umbenannt. Wenn wir die Tabelle **Anreden** mit den beiden Feldern **AnredeID** und **Anrede** als Datenquelle einer Entity Framework-Klasse nutzen wollen, würde das Feld **Anrede** automatisch in die Eigenschaft **Anrede1** umbenannt, da die Klasse dann logischerweise **Anrede** heißen soll. Auch der Versuch, diese Eigenschaft im Entity Framework umzubenennen, scheitert mit der Meldung aus Bild 9.

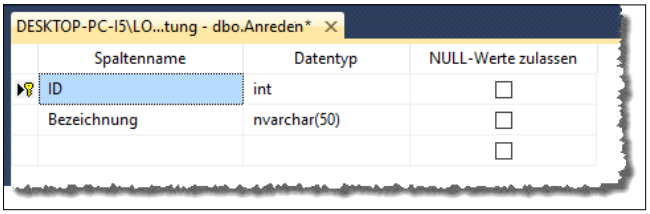


Bild 8: Definierter Primärschlüssel

Welche Schlussfolgerung ziehen wir daraus? Wir verwenden statt der Bezeichnung **Anrede**, die mit dem Entitätsnamen übereinstimmt, eine andere Bezeichnung. In diesem Fall weichen wir auf den Feldnamen **Bezeichnung** aus.

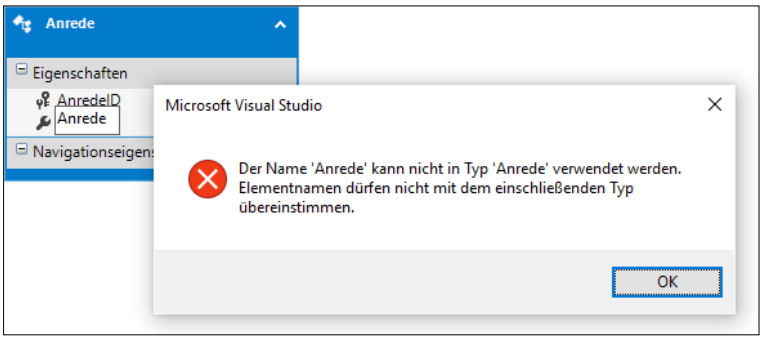


Bild 9: Im Entity Framework dürfen Eigenschaften und Klassen nicht den gleichen Namen verwenden.

Wir haben zwar bereits geeignete Datentypen für die beiden Felder ausgewählt, aber wenn sie von Access kommen, dürfte es Sie interessieren, mit welchen Datentypen unter SQL Server man die Datentypen unter Access ersetzen kann. Daher finden Sie nachfolgende eine Auflistung der passenden Datentypen für Access und SQL Server.

Vor dem Doppelpunkt einer jeden Zeile finden Sie den Datentyp unter Microsoft Access, hinter dem Doppelpunkt haben wir den Datentyp im SQL Server aufgeführt:

- **Autowert** (**Long Integer**): **integer** mit Spaltenoption **IDENTITY**
- **Autowert** (**Replikations-ID**): **uniqueidentifier** mit der Funktion **NEWID()** als Standardwert
- **Zahl** (**Byte**): **tinyint**
- **Zahl** (**Integer**): **smallint**
- **Zahl** (**Long Integer**): **integer**
- **Zahl** (**Single**): **real**
- **Zahl** (**Double**): **float**
- **Zahl** (**Dezimal**): **decimal** oder **numeric**
- **Zahl** (**Replikations-ID**): **uniqueidentifier**
- **Währung**: **money**
- **Ja/Nein**: **bit** mit Option **NOT NULL**
- **Text**: **nvarchar(n)** für Unicode und **varchar(n)** für ANSI; **(n)** steht für die maximale Länge
- **Memo**: **ntext** oder **nvarchar(max)** für Unicode und **text** oder **varchar(max)** für ANSI
- **Hyperlink**: **ntext** oder **nvarchar(max)**, jedoch ohne Hyperlink-Funktionalität
- **Datum/Uhrzeit**: **datetime**
- **OLE-Objekt**: **image** oder **varbinary(max)**
- **Anlage**: **ntext** oder **nvarchar(max)**
- **Text** als **RichText**: **ntext** oder **nvarchar(max)** inklusive HTML-Tags

Zusammengefasst verwenden wir für die Tabelle **Anreden** für das Feld **ID** ein **integer**-Feld mit der oben bereits beschriebenen **IDENTITY**-Eigenschaft und der Primärschlüssel-Eigenschaft. Das Feld Bezeichnung statten wir mit dem Datentyp **nvarchar(50)** aus. Außerdem belassen wir für beide Felder die Eigenschaft **NULL-Werte zulassen** bei dem Wert **Nein**, denn beide Felder müssen für jeden Datensatz unbedingt gefüllt werden.

Datensätze anlegen

Bevor wir Datensätze über eine entsprechende Benutzeroberfläche hinzufügen können, wird es noch ein paar

SQL Server-Interaktion mit dem Profiler verfolgen

Wenn Sie per ADO.NET, LINQ to Entities oder anderen Zugriffstechniken auf die Datenbank eines SQL Servers zugreifen, werden die von Ihnen programmierten Codezeilen meist im Hintergrund in SQL-Befehle umgewandelt. In vielen Fällen kann es interessant sein, was dort tatsächlich geschieht. Ein geeignetes Werkzeug ist der SQL Server Profiler, der zwar demnächst ausläuft, aber eine schnelle Möglichkeit der Nachverfolgung liefert.

Wir wollen in diesem Artikel die schnelle und unkomplizierte Verfolgung der Zugriffe auf den SQL Server beschreiben. Dazu ist eine Installation des SQL Server Profilers in einer aktuellen und dem SQL Server entsprechenden Version Voraussetzung. Nachdem Sie den SQL Server Profiler gestartet haben, erwartet Sie zunächst das leere Anwendungsfenster (siehe Bild 1). Hier klicken Sie direkt auf die Schaltfläche **Neue Ablaufverfolgung**.

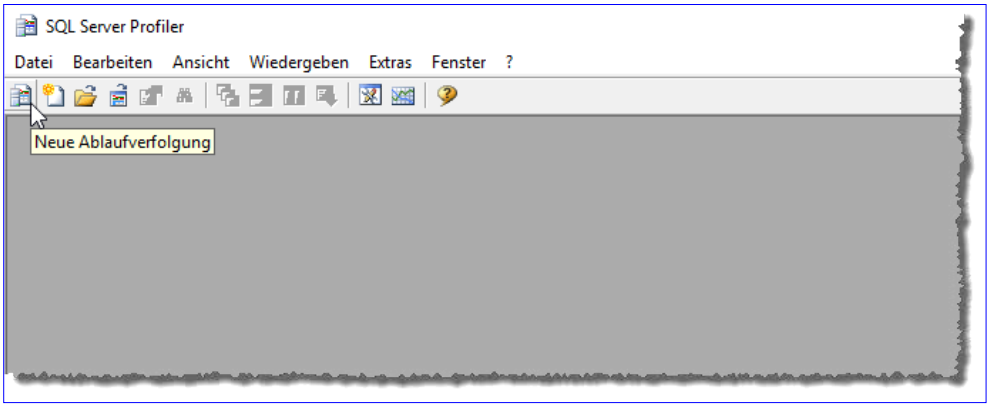


Bild 1: Der Profiler nach dem Start der Anwendung

Daraufhin erscheint der Dialog aus Bild 2, mit dem Sie sich an der SQL Server-Instanz anmelden, die Sie untersuchen möchten. In diesem Fall geht es um eine Instanz von LocalDB in der Version 11.0. Nach der Auswahl/Eingabe des Servernamens klicken Sie auf die Schaltfläche **Verbinden**.

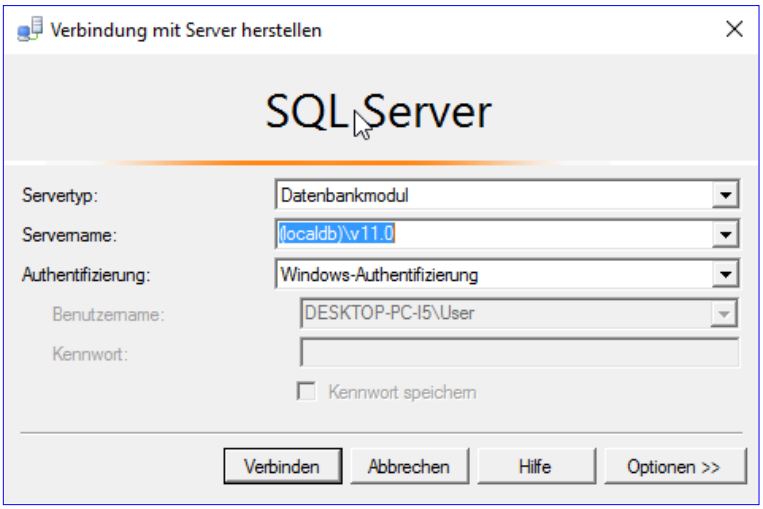


Bild 2: Angabe des zu untersuchenden Servers

Damit erscheint der Dialog **Ablaufverfolgungseigenschaften**, der einige Einstellungen ermöglicht (siehe Bild 3). Die erste ist die Vergabe eines Namens für die Ablaufverfolgung. Außerdem stellen Sie ein, ob die protokollierten Daten in einer Datei oder in einer SQL Server-Tabelle gespeichert werden sollen. Sollten Sie letztere Variante wählen und wollen Sie damit die Performance untersuchen, sollten Sie eine Tabelle einer Datenbank verwenden, die von einer anderen SQL Server-Instanz verwaltet wird.

Dazu öffnet der Profiler, sobald Sie die Option **In Tabelle speichern** gewählt haben, direkt einen weiteren Dialog zum Herstellen einer Verbindung mit einem SQL Server, sodass Sie die Gelegenheit haben, auch eine alternative Instanz auszuwählen.

Die SQL-Anweisungen zum Speichern der Profiler-Daten werden übrigens nicht mitgespeichert.

Das Speichern in einer SQL Server-Tabelle ist dann notwendig, wenn Sie viele Daten oder einen längeren Zeitraum beobachten wollen – die Darstellung im Profiler selbst ist nicht dazu geeignet, beispielsweise nach bestimmten Tabellen oder Ereignissen zu filtern. Für die im Rahmen des Zugriffes durch die in diesem Magazin beschriebenen Beispiele reicht die Darstellung direkt im Profiler allerdings aus.

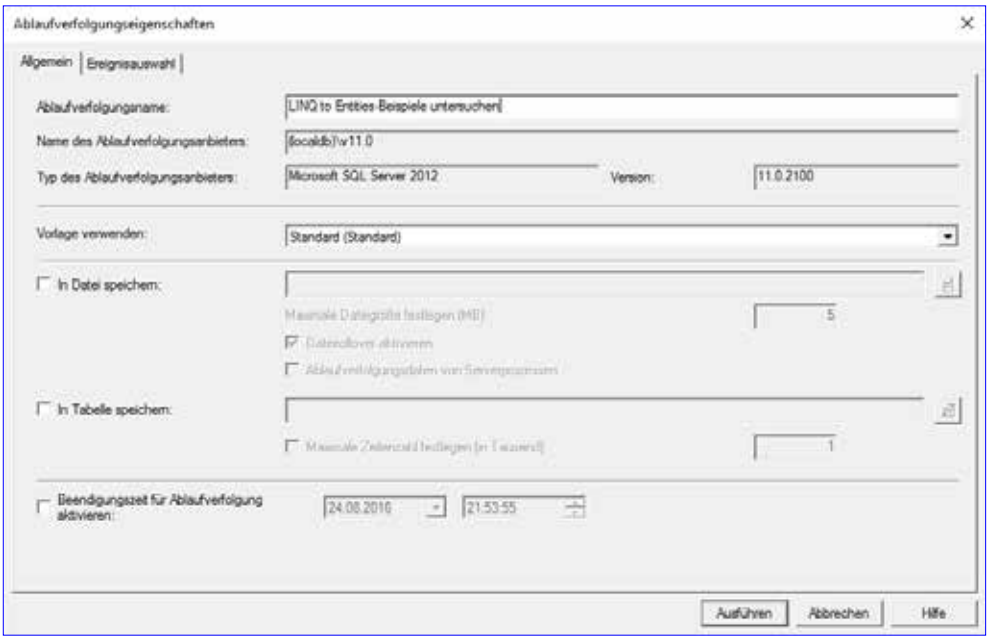


Bild 3: Eingabe der Ablaufverfolgungseigenschaften

Nach Datenbank filtern

Interessant wird es, wenn Sie die zu untersuchenden Zugriffe auf die Datenbank auf einer Server-Instanz ausführen, die noch andere, gegebenenfalls rege frequentierte Datenbanken verwaltet. In diesem Fall müssen Sie noch festlegen, dass der Profiler nur die Zugriffe auf die von Ihnen verwendete Datenbank protokolliert. Dies erledigen Sie auf der zweiten Registerseite des Dialogs **Ablaufverfolgungseigenschaften** (siehe Bild 4).

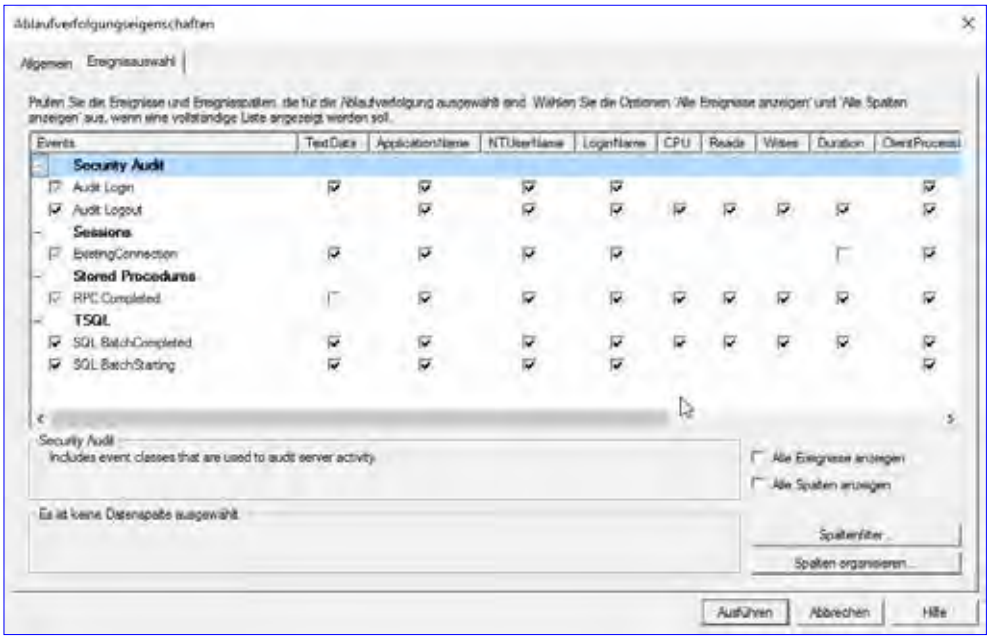


Bild 4: Festlegen der zu protokollierenden Ereignisse

Diese Registerseite enthält eine Matrix aus Ereignissen und den für die jeweiligen Ereignisse aufgezeichneten Daten. Wenn Sie festlegen möchten, dass nur die Zugriffe auf eine bestimmte Datenbank protokolliert werden sollen, müssen Sie zunächst einmal die ID dieser Datenbank herausfinden und einen entsprechenden Filter definieren.

Dazu öffnen Sie das SQL Server Management Studio und verbinden sich mit dem

Entity Data Model für eine Datenbank erstellen

In den vorangegangenen Ausgaben von DATENBANKENTWICKLER haben Sie bereits erfahren, wie Sie per ADO.NET auf die Daten der Tabellen einer Datenbank zugreifen. Nun nutzen wir nicht mehr direkt ADO.NET, sondern das Entity Framework als Datenlieferant, welches eine ganz andere Art des Zugriffs ermöglicht. Dieser Artikel zeigt zunächst, wie Sie die Verbindung zu einer Beispieldatenbank einrichten und dann die für den Zugriff per C# oder die Anzeige in WPF-Fenstern und -Steuerelementen nötigen Objekte erstellen.

Im Artikel **Datenbanken erstellen mit dem SSMA** haben wir eine kleine Beispieldatenbank mit dem SQL Server Management Studio auf Basis von **LocalDB** erstellt. Auf diese wollen wir nun mit dem Entity Framework zugreifen, um die Vorbereitung für den Zugriff per C# oder die Anzeige in mit WPF definierten Benutzeroberflächen zu erledigen.

Bild 1: Hinzufügen eines neuen Entity Data Models

Entity Data Model anlegen

Als Erstes erstellen wir dazu das Entity Data Model auf Basis der Tabellen der Beispieldatenbank. Dazu legen Sie ein neues Projekt des Typs **C#|Konsolenanwendung** an und nennen es **EF_Zugriff**.

Wählen Sie dazu im Objektmappen-Explorer auf dem Kontextmenü des Projekts den Eintrag **Hinzufügen|Neues Element** aus. Der Typ des gewünschten neuen Objekts heißt **ADO.NET Entity Data Model**, der Name soll **BestellverwaltungModel** lauten (siehe Bild 1).

Im nun erscheinenden Dialog **Assistent für Entity Data Model** wählen Sie unter dem Schritt **Modellinhalte auswählen** den Eintrag **EF Designer aus Datenbank** aus, da wir ja bereits eine Datenbank vorbereitet haben (siehe Bild 2).

Daraufhin erscheint der Dialog-Schritt **Wählen Sie Ihre Datenverbindung aus**. Hier klicken Sie auf **Neue Verbindung...** und wählen im folgenden Dialog **Datenquelle auswählen** den Eintrag **Microsoft SQL Server** aus (siehe Bild 3).

Bild 2: Art des Modellinhalts

Tragen Sie dort den Namen des Servers ein oder wählen Sie ihn aus der Liste aus, soweit vorhanden. Wenn Sie, wie oben vorgeschlagen, LocalDB verwenden und eine einzige Instanz von LocalDB auf dem Rechner installiert haben, können Sie hier **(localdb)\SQLLocalDB** eintragen, sonst geben Sie die Version mit an – also etwa **(localdb)\v11.0** wie in Bild 4. Wählen Sie dann unter **Mit Datenbank verbinden** die gewünschte Datenbank aus, sofern diese schon von LocalDB verwaltet wird. Anderenfalls können Sie eine **.mdf**-Datei unter **Datenbankdatei anhängen** angeben. Klicken Sie auf **OK**.

Damit kehren Sie zum Schritt **Wählen Sie Ihre Datenverbindung aus** und können dort mit einem Klick auf die Schaltfläche **Weiter >** zum nächsten Schritt wechseln.

Der Assistent fragt Sie nun, ob Sie, falls Sie eine Datenbankdatei angegeben haben, diese in den Projektordner kopieren möchten. Dies macht beispielsweise Sinn, wenn Sie diese Datei mit dem Projekt weitergeben möchten.

Im nächsten Schritt fragt der Assistent Sie unter dem Titel **Wählen Sie Ihre Version**, welche Version des Frameworks

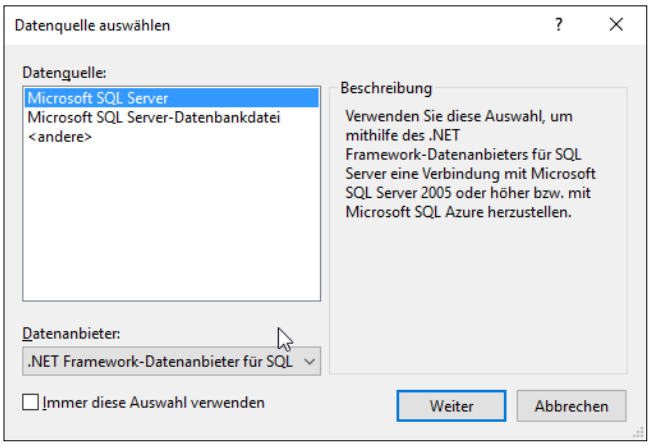


Bild 3: Auswahl der Datenquelle

Sie verwenden möchten. Wir wählen den Wert **Entity Framework 6.x** aus.

Im nächsten Schritt wird es nun interessant: Hier finden wir die Objekte der Datenbank vor, in diesem Fall die wie im Artikel **Datenbanken erstellen mit dem SSMA** angelegten Tabellen (siehe Bild 5). Um alle hier angegebenen Tabellen gleichzeitig zu markieren, klicken Sie einfach auf den Eintrag **Tabellen**. Unten finden Sie noch zwei Optionen, mit denen Sie die Erstellung der Klassen des Entity Frameworks steuern

Bild 4: Auswahl der Datenbank

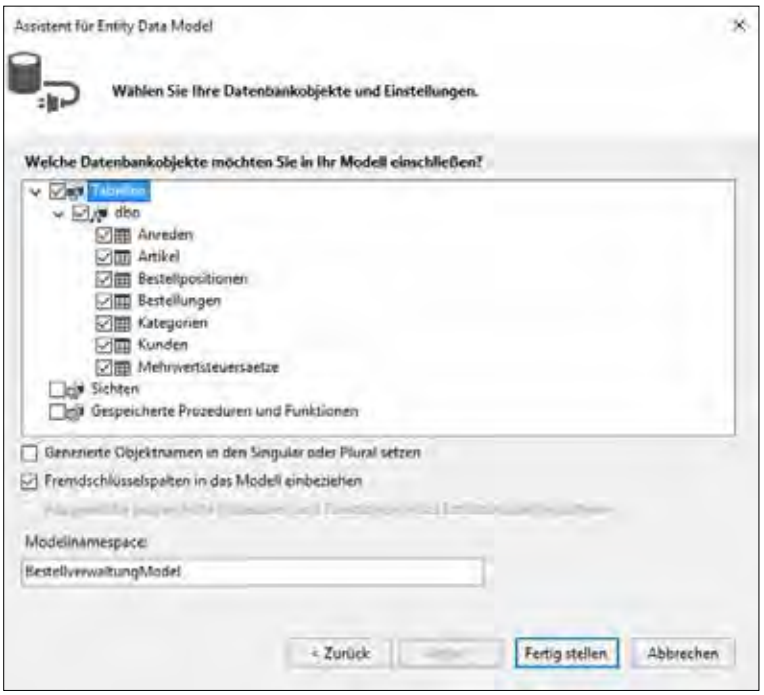


Bild 5: Auswahl der zu verwendenden Datenbankobjekte

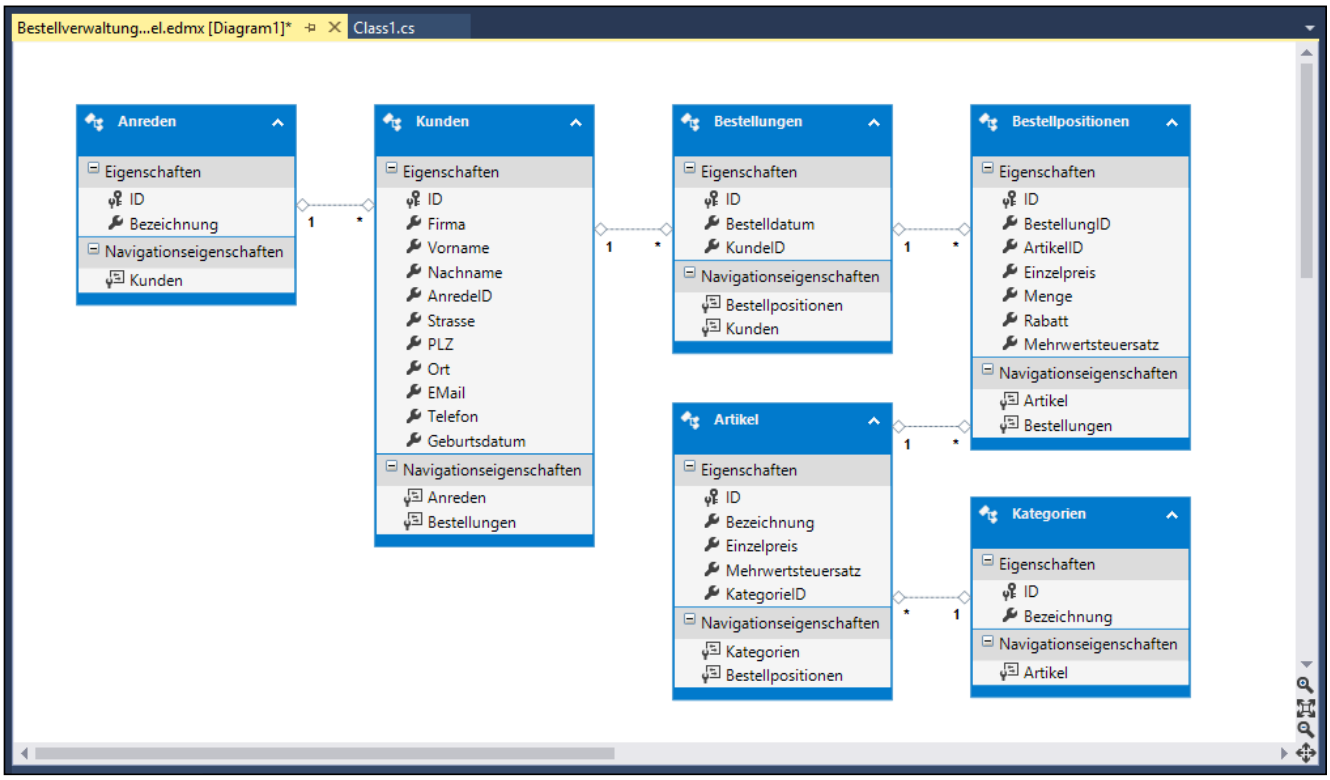


Bild 6: Entity Data Model der Bestellverwaltung

können. Die erste ist für uns nicht relevant, da wir den Tabellen des Datenmodells deutsche Bezeichnungen gegeben haben – bei englischen Bezeichnungen würde einem diese Option etwas Arbeit durch das Anhängen/Entfernen von s bei Singular/Plural abnehmen (**Customer**/**Customers**). Die Option **Fremdschlüsselspalten in das Modell einbeziehen** behalten wir hingegen bei. Außerdem legen wir als Name des Namespaces für das Model den vorgegebenen Wert **BestellverwaltungModel** fest.

Damit können Sie nun auf **Fertig stellen** klicken. Das Ergebnis finden Sie nach einer kurzen Zeit in Bild 6. Die Abbildung zeigt das Entity Data Model auf Basis der angegebenen Tabellen samt ihrer Verknüpfungen beziehungsweise Fremdschlüsselfelder. Die Ansicht sieht etwas anders aus als das Datenmodell, das Sie etwa vom Beziehungsfenster von Access kennen. Hier werden die Beziehungspfeile nicht direkt mit den betroffenen Feldern der verknüpften Tabellen verbunden, sondern einfach an irgendeiner Stelle. Dafür zeigen die einzelnen Entitäten jeweils die verknüpften Elemente unten im Bereich **Navigationseigenschaften** an. Die Felder selbst werden nun **Eigenschaften** genannt.

Entitäten: Aus Plural mach Singular

An dieser Stelle sehen Sie, dass die Entitäten des Entity Data Models genauso heißen wie die Tabellen, von denen sie abgeleitet wurden. Schön und gut – aber wenn wir später im Code auf diese Entitäten zugreifen wollen, was letztlich Klassen sind, auf deren Basis sie Objekte erstellen, dann fällt eines auf: Ein Objekt etwa zur Abbildung eines Kunden soll wohl kaum beispielsweise **Kunden** heißen, sondern **Kunde**, und Gleiches gilt dann auch für die Klassen (nur dass wir Objekte, also Instanzen von Klassen, nicht genau wie die Klassen benennen, sondern diese beispielsweise mit einem kleinen Anfangsbuchstaben versehen – also **kunde** oder **bestellung**).

Nun sollten wir also dafür sorgen, dass die Klassen nicht mehr **Kunden** oder **Bestellungen** heißen, sondern **Kunde** oder **Bestellung**. Dazu benennen Sie gleich hier und jetzt

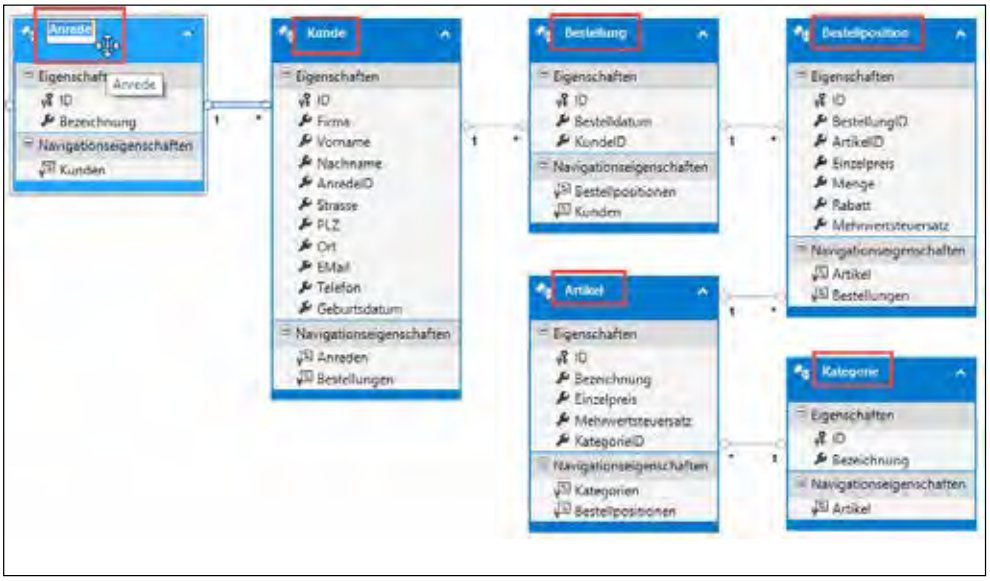


Bild 7: Entity Data Model mit Singular-Namen für die Entitäten

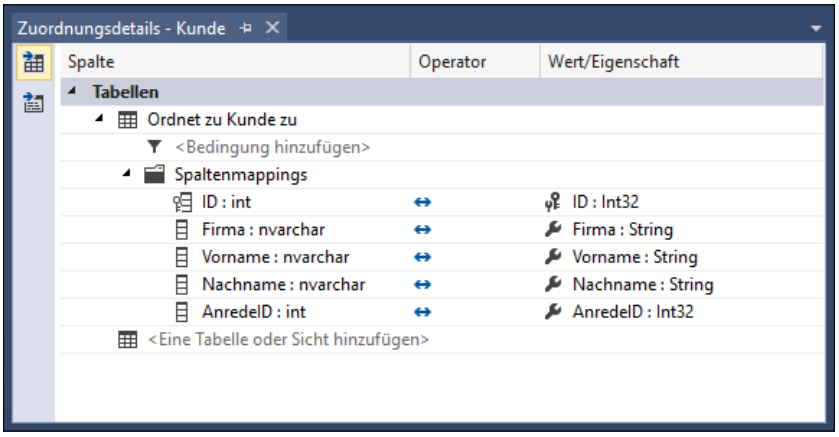


Bild 8: Mapping der Eigenschaften der Klassen zu den Tabellenfeldern

die Bezeichnungen im Entity Data Model um, indem Sie die betroffene Klasse anklicken, nochmals auf den Namen im Titel klicken und diesen dann ändern. Aus den bisherigen Bezeichnungen wird dann **Kategorie**, **Artikel**, **Bestellposition**, **Bestellung**, **Kunde**, und **Anrede** (siehe Bild 7). Speichern Sie dann das Entity Data Model, damit die Änderungen auch auf die Klassen des Entity Data Models übertragen werden,

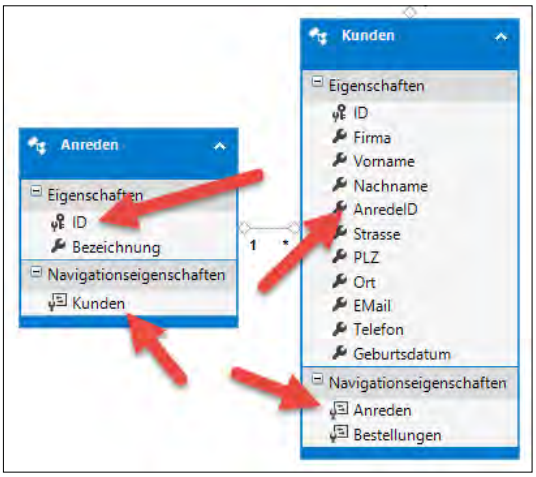


Bild 9: Markierung der an einer Beziehung beteiligten Eigenschaften

welche diese Bezeichnungen enthalten.

Mapping von Entität zu Tabelle

Wenn Sie auf eine der Entitäten klicken, erscheint unterhalb im Bereich **Zuordnungsdetails** das Mapping, also die Angabe, welche Eigenschaft zu welchem Feld der zugrunde liegenden Tabelle gehört. Hier finden Sie auch gleich die Datentypen auf Datenbankseite und Objektseite vor (siehe Bild 8).

Um Informationen über die Umsetzung der Beziehungen zwischen den zugrunde liegenden Tabellen zu erhalten, klicken Sie auf die jeweilige Navigationseigenschaft. Wenn Sie etwa erfahren möchten, welche Felder an der Beziehung zwischen den Entitäten **Anrede** und **Kunde** beteiligt sind, klicken Sie auf die Navigationseigenschaft **Kunde** der Entität **Anrede** oder die Navigationseigenschaft **Anrede** der Entität **Kunde**. Die beteiligten Eigenschaften beziehungsweise Navigationseigenschaften werden dann unterstrichen hervorgehoben (siehe Bild 9).

Achtung: An dieser Stelle macht sich bemerkbar, dass wir – auch wegen der Verwendung deutscher Namen für die Tabellen/Entitäten – die Option **Generierte Objektnamen in den Singular oder Plural setzen** nicht aktiviert haben. Anderenfalls würde die Navigati-

LINQ to Entities: Daten abfragen

In den vorangegangenen Ausgaben von DATENBANKENTWICKLER haben Sie bereits erfahren, wie Sie per ADO.NET auf die Daten der Tabellen einer Datenbank zugreifen. Nun nutzen wir nicht mehr direkt ADO.NET, sondern das Entity Framework als Datenlieferant, welches eine ganz andere Art des Zugriffs ermöglicht. Dieser Artikel zeigt, wie Sie per C#-Konsolenanwendung auf die per Entity Data Model bereitgestellten Daten zugreifen.

Beispielprojekt

Das Beispielprojekt soll über ein geeignetes Entity Data Model auf die Tabellen einer LocalDB-Datenbank zugreifen. Wie Sie das Entity Data Model erstellen, beschreiben wir im Artikel **Entity Data Model für eine Datenbank erstellen**.

Alle Beispielmethoden finden Sie in der Klasse **EDM_ZugriffPerCSharp_Beispiele.cs**. Um die Beispiele auszuprobieren, starten Sie einfach das Projekt.

Aufruf der Beispielmethoden

Damit wir die nachfolgend programmierten Methoden einfach aufrufen können, verwenden wir wieder die bereits in früheren Artikeln verwendete Klasse **Program** mit einer **Main**-Methode, die alle öffentlichen, statischen Methoden aus öffentlichen Klassen auflistet und zur Ausführung anbietet. In einer neuen Klasse namens **EDM_ZugriffPerCSharp_Beispiele.cs** legen wir dann die später vorgestellten Methoden an.

LINQ to Entities

Die Abfragesprache für den Zugriff auf die Daten des Entity Frameworks und damit auf das Entity Data Model heißt **LINQ to Entities**.

Zugriff auf die Daten der Tabelle Anrede

Unter Access würden Sie nun, wenn Sie alle Datensätze einer Tabelle wie etwa Anrede durchlaufen wollten, ein **Database**-Objekt und ein **Recordset**-Objekt erzeugen, Letzteres mit einer entsprechenden SQL-Anweisung füllen und dann per VBA in einer **Do While**-Schleife durch die einzelnen Datensätze navigieren. Unter C# und dem Entity Framework sieht das etwas anders aus – im Überblick wie in Listing 1. Hier nutzen Sie ein automatisch generiertes Objekt, das die Klasse **DbContext** implementiert, die in unserem Fall beispielsweise **BestellverwaltungEntities** heißt.

Wir erstellen ein neues Objekt auf Basis dieses Typs und speichern es in der Variablen **context**:

```
BestellverwaltungEntities context =
                        new BestellverwaltungEntities();
```

Dann definieren wir eine Variable des Typs **var** und nennen diese **anreden**. Sie soll mit der Liste der Anreden des Objekts **context** gefüllt werden:

```
var anreden = context.Anreden;
```

Schließlich durchlaufen wir die Elemente der Auflistung **anreden** in einer **foreach**-Schleife und weisen das jeweils aktuelle Objekt der Variablen **anrede** zu. Innerhalb der

```
public static void AlleAnredenAusgeben_ForEach() {
    BestellverwaltungEntities context = new BestellverwaltungEntities();
    var anreden = context.Anreden;
    foreach (var anrede in anreden) {
        Console.WriteLine("{0} {1}", anrede.ID, anrede.Bezeichnung);
    }
}
```

Listing 1: Methode zur Ausgabe aller Anreden der Beispieldatenbank

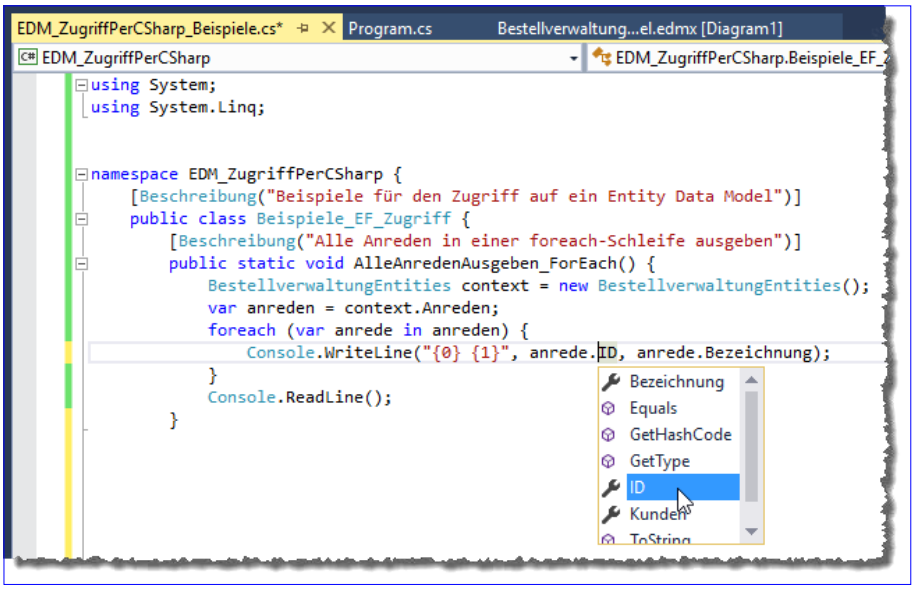


Bild 1: IntelliSense für den Datenzugriff

Schleife geben wir dann die Werte der Eigenschaften **ID** und **Bezeichnung** aller Elemente aus:

```
foreach (var anrede in anreden) {
    Console.WriteLine("{0} {1}",
                      anrede.ID, anrede.Bezeichnung);
}
```

Das Ergebnis sieht etwa so aus:

```
1 Herr
2 Frau
```

Praktischerweise blendet Visual Studio beim Eingeben der Eigenschaften für das Objekt **anrede** per IntelliSense die verfügbaren Einträge ein (siehe Bild 1).

Fertig – damit wäre der erste Zugriff auf die Daten der Tabelle **Anreden** unserer Beispieldatenbank schon erledigt! Im Folgenden sehen wir uns nun weitere Möglichkeiten an.

Verbesserung: using verwenden

Damit die Verbindung zur Datenbank nach dem Abrufen der Daten wieder geschlossen wird, fassen Sie die für den Datenzugriff verwendeten Methoden in ein **using**-Konstrukt ein. Der Grund, ohne weitere Erläuterungen: Die **DbContext**-Klasse, von der unsere Klasse erbt, implementiert das Interface **IDisposable**. Die Nutzung solcher Objekte sollte man innerhalb der **using**-Direktive realisieren. Auf diese Weise wird beim Verlassen des von der **using**-Direktiven eingefassten Abschnitts auf jeden Fall die **Dispose**-Methode aufgerufen, was zum Beispiel die genutzten Ressourcen wieder freigibt. Die verbesserte Version des vorherigen Beispiels sieht nun wie in Listing 2 aus.

Achtung: Die **using**-Variante funktioniert nicht wie gewünscht, wenn Sie die der **using**-Direktiven übergebene Variable bereits vorher deklarieren, wie es hier der Fall ist. Die Variable existiert dann auch noch nach Verlassen von **using**:

```
BestellverwaltungEntities context =
                                new BestellverwaltungEntities()
using (context) {
    ... }
```

```
public static void AlleAnredenAusgeben_ForEach_Besser() {
    using (BestellverwaltungEntities context = new BestellverwaltungEntities()) {
        var anreden = context.Anreden;
        foreach (var anrede in anreden) {
            Console.WriteLine("{0} {1}", anrede.ID, anrede.Bezeichnung);
        }
    }
    Console.ReadLine();
}
```

Listing 2: Methode zur Ausgabe aller Anreden der Beispieldatenbank mit **using**

Kunden sortiert ausgeben

Wenn Sie die Daten einer Tabelle sortiert erhalten möchten, können Sie das durch eine Erweiterung einer Abfrage erreichen. Alle Kunden in unsortierter Reihenfolge würde folgendes Statement liefern:

```
var kunden = context.Kunden;
```

Wenn Sie eine Sortierung hinzufügen wollen, etwa nach dem Feld **Nachname**, haben Sie zwei Möglichkeiten – eine mit der Methodensyntax und eine mit der Abfragesyntax.

Abfragen mit der Methodensyntax

Für beide fügen Sie zunächst eine Referenz auf den Namespace **System.Linq** zur Klassendatei hinzu:

```
using System.Linq;
```

Nun erstellen Sie eine Methode wie in Listing 3. Der Unterschied zur Ermittlung der unsortierten Kunden liegt in der folgenden Zeile:

```
var kunden = context.Kunden.OrderBy(d => d.Nachname);
```

Hier haben wir noch die Methode **OrderBy** an die Eigenschaft **Kunden** angehängt und dieser als Parameter die Vorgabe für die Sortierung mitgegeben, nämlich **d => d.Nachname**.

Dabei handelt es sich um einen sogenannten Lambda-Ausdruck. Lambda-Ausdrücke haben wir in diesem Magazin noch nicht besprochen, daher gehen wir hier nur darauf ein, wie diese für den vorliegenden Anwendungszweck formuliert werden.

Der Lambda-Ausdruck, den wir hier als Parameter der **OrderBy** (und später auch der **OrderByDescending**-, der **Where**- und anderer Methoden) nutzen, enthält zwei Teile: Der Teil vor dem Lambda-Zeichen (**=>**, nicht zu verwechseln mit **=>**) enthält den Namen für das, was wir im Teil hinter dem Lambda-Zeichen untersuchen wollen.

Wir untersuchen immer das Objekt, für das wir die Methode **OrderBy** (**OrderByDescending**, **Where**, ...) aufrufen, in diesem Fall also **context.Kunde**, also die Auflistung der Kunden-Entitäten, und geben dieser im vorliegenden Fall den Namen **d** (Sie können auch einen anderen Buchstaben oder Ausdruck statt **d** verwenden).

Dies können wir dann im zweiten Teil des Lambda-Ausdrucks aufgreifen. Im Falle der **OrderBy**-Methode geben wir also beispielsweise an, wonach wir sortieren wollen – hier die Eigenschaft **Nachname** der **Kunde**-Entitäten.

In der Variablen **kunden** landet dann die sortierte Liste der **Kunde**-Entitäten, die wir wieder in einer **foreach**-Schleife durchlaufen. Diesmal geben wir den Primärschlüsselwert sowie den Nachnamen der Kunden aus:

```
8 Ackermann
9 Alberts
10 Becker
11 Fröhlich
4 Meier
1 Minhorst
2 Müller
7 Schmidt
```

```
public static void KundenNachNachname_MethodenSyntax() {
    using (BestellverwaltungEntities context = new BestellverwaltungEntities()) {
        var kunden = context.Kunden.OrderBy(d => d.Nachname);
        foreach (var kunde in kunden) {
            Console.WriteLine("{0} {1}", kunde.ID, kunde.Nachname);
        }
    }
    Console.ReadLine();
}
```

Listing 3: Methode zur Ausgabe der Kunden, sortiert nach dem Nachnamen (Methodensyntax)

Abfragen mit der Abfragesyntax

Die zweite Methode zum Abfragen von Daten unter LINQ to

Entities verwendet die sogenannte Abfragesyntax. Hier stellen Sie einen Abfrageausdruck zusammen, der eher wie ein SQL-Ausdruck aussieht. Um das gleiche Ergebnis wie im obigen Beispiel mit der Methodensyntax zu erzielen, ersetzen Sie die Zeile mit der **OrderBy**-Methode und dem Lambda-Ausdruck durch die folgende Zeile:

```
var kunden =
from d in context.Kunden
orderby d.Nachname
select d;
```

Dies sieht für erfahrene SQL-Nutzer etwas intuitiver aus, da hier einige Elemente wie **from**, **orderby** und **select** auftauchen. Die Reihenfolge ist hier etwas anders als beim Lambda-Ausdruck. Die Zuweisung des Ergebnisses erfolgt wieder an die Variable **kunden**. Danach geben wir an, woher die Daten kommen und welchen Alias-Namen wir nachfolgend dafür verwenden wollen (**from d in context.Kunden**).

Dann legen wir die Operation fest, in diesem Fall wieder die Sortierung (**orderby d.Nachname**). Schließlich gibt **select d** an, was zurückgegeben werden soll. Beim vollständigen Beispiel ändert sich also nur die Zeile mit der Definition der Abfrage (siehe Listing 4).

```
public static void KundenNachNachname_Abfragesyntax() {
    using (BestellverwaltungEntities context = new BestellverwaltungEntities()) {
        var kunden = from d in context.Kunden orderby d.Nachname select d;
        foreach (var kunde in kunden) {
            Console.WriteLine("{0} {1}", kunde.ID, kunde.Nachname);
        }
    }
    Console.ReadLine();
}
```

Listing 4: Methode zur Ausgabe der Kunden, sortiert nach dem Nachnamen (Abfragesyntax)

Sowohl bei der Methodensyntax als auch bei der Abfragensyntax unterstützt Visual Studio Sie mit IntelliSense.

Absteigende Sortierung

Unter SQL haben Sie für eine absteigende Sortierung das Schlüsselwort **DESC** hinten angefügt (**ORDER BY Nachname DESC**). Unter LINQ to Entities verwenden Sie bei der Methodensyntax einfach eine andere Methode (**OrderByDescending**):

```
var kunden =
context.Kunden.OrderByDescending(d => d.Nachname);
```

Bei der Abfragensyntax fügen Sie ähnlich wie bei SQL ein Schlüsselwort hinzu, hier **descending**:

```
var kunden =
from d in context.Kunden
orderby d.Nachname descending
select d;
```

Hinweis: Die Abfrage wird zwar in den Beispielen bereits vor der Schleife definiert, aber die Daten werden erst beim ersten Zugriff in der Schleife geladen.

Kunden nach bestimmten Kriterien ausgeben

Nun steigern wir den Schwierigkeitsgrad und wollen nur bestimmten Datensätze der Tabelle **Kunden** als Objekt ermitteln und durchlaufen. Unter Access/VBA brauchen wir dazu nur die **SELECT**-Anweisung in der **OpenRecordset**-Methode zu ändern. Unter C#/LINQ to Entities ist es auch nicht viel komplizierter – genau genommen haben Sie die Grundlagen dazu schon oben mit der Sortierung kennen gelernt.

Wenn Sie beispielsweise den Kunden ermitteln wollen, dessen Eigenschaft **Nachname** den Wert **Minhorst** enthält, formulieren Sie die Abfrage in der Methodensyntax wie folgt (siehe Methode **KundeMitNachname_Methodensyntax**).

Sie verwenden also die **Where**-Methode statt der **OrderBy**-Methode:

```
var kunden =
context.Kunden.Where(d => d.Nachname == "Minhorst");
```

In der Abfragensyntax erledigen Sie die Aufgabe mit dieser Zeile (siehe Methode **KundeMitNachname_Abfragesyntax**). Hier kommt also das **where**-Schlüsselwort mit dem Vergleichskriterium hinzu:

```
var kunden =
from d in context.Kunden
where d.Nachname == "Minhorst"
select d;
```

Platzhalter?

Von Access/SQL kennen Sie als Platzhalter das Sternchen (*) für kein, ein oder mehrere Zeichen oder das Fragezeichen für ein Zeichen, vom SQL Server das Prozentzeichen (%). Unter LINQ to Entities gibt es Platzhalter in diesem Sinne nicht.

Auf einem Umweg können Sie auch damit arbeiten, aber dazu kommen wir gegebenenfalls in einem späteren Artikel.

Für die meisten Fälle sollten jedoch einige spezielle Funktionen ausreichen, etwa solche, mit denen Sie nach Inhalten suchen können, die mit einem bestimmten Ausdruck beginnen, einen Ausdruck enthalten oder damit enden.

Kunden, deren Nachname mit einem bestimmten Ausdruck beginnt

Suchen Sie nach Objekten, von denen ein Eigenschaftswert mit einem bestimmten Ausdruck beginnt – etwa der Nachname mit dem Buchstaben **A** –, dann nutzen Sie eine erweiterte Variante der **Where**-Methode beziehungsweise des **Where**-Schlüsselworts.

Bei der Methodensyntax nutzen Sie dann nicht etwa einen Ausdruck wie **d => d.Nachname == "A%"**, denn Platzhalter gibt es in dieser Form ja nicht. Stattdessen nutzen Sie die Funktion **StartsWith** der entsprechenden Eigenschaft und übergeben dieser den Wert, mit dem der Nachname der zu liefernden Objekte beginnen soll (**d => d.Nachname.StartsWith("A")**), siehe Methode **KundenMitNachnameBeginntMitA_Methodensyntax**:

```
var kunden =
context.Kunden.Where(d => d.Nachname.StartsWith("A"));
```

Für die Abfragensyntax sieht das wie folgt aus – hier nutzen wir die Funktion **StartsWith** analog zur Methodensyntax (siehe Methode **KundenMitNachnameBeginntMitA_Abfragesyntax** in der Beispieldatenbank):

```
var kunden =
from d in context.Kunden
where d.Nachname.StartsWith("A") select d;
```

Die verfügbaren Funktionen wie **StartsWith**, **EndsWith**, **Contains** und viele andere werden per IntelliSense angezeigt, nachdem Sie den Punkt hinter dem Objekt angegeben haben (siehe Bild 2).

Andere Zeichenketten-Funktionen

Es gibt noch eine Reihe weiterer Funktionen, die Sie für eine Eigenschaft angeben können, um diese nach einer enthaltenen Zeichenkette zu durchsuchen – zum Beispiel die folgenden beiden:

- **contains**: Liefert alle Elemente, welche die angegebene Zeichenkette enthalten.

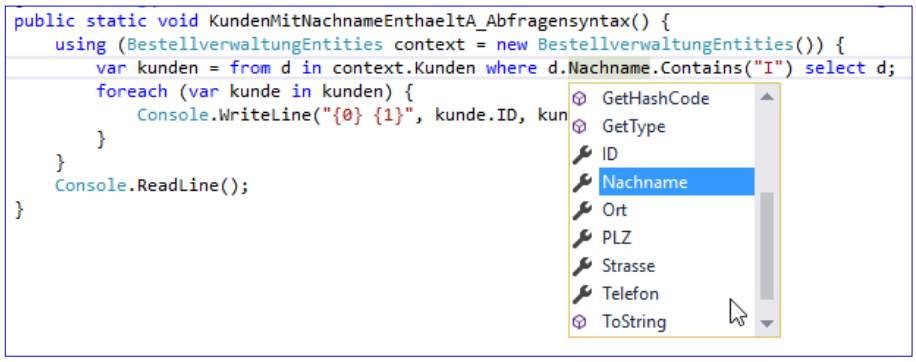


Bild 2: Auswahl der möglichen Funktionen per IntelliSense

- **endswith**: Sucht die Elemente, die mit der angegebenen Zeichenkette enden.

Sortieren und Filtern gleichzeitig

Gelegentlich wollen Sie gleichzeitig filtern und sortieren oder auch mehrere Filter- oder Sortierkriterien übergeben. Alle Kunden, deren Name ein **i** enthält, sortiert der folgende Ausdruck in der Abfragensyntax auch direkt nach dem Nachnamen:

```
var kunden =
from d in context.Kunden
where d.Nachname.Contains("I")
orderby d.Nachname
select d;
```

In der Methodensyntax hängen Sie einfach die beiden Methoden hintereinander an die zu untersuchende/zu sortierende Eigenschaft an:

```
var kunden = context.Kunden
.Where(d => d.Nachname.Contains("i"))
.OrderBy(d => d.Nachname);
```

Nach mehreren Felder filtern

Auf die gleiche Weise können Sie auch nach mehreren Feldern gleichzeitig sortieren oder filtern. Hängen Sie einfach die gewünschten Kriterien hintereinander.

Alle Kunden, deren **Nachname** mit **A** beginnt und deren **AnredeID** den Wert **1** enthält, liefert die folgende Zeile in der Abfragensyntax:

```
var kunden =
from d in context.Kunden
where d.Nachname.StartsWith("A")
where d.AnredeID == 1
orderby d.Nachname
select d;
```

Und hier der Ausdruck in der Methodensyntax:

```
var kunden = context.Kunden
.Where(d => d.Nachname.StartsWith("A"))
.Where(d => d.AnredeID == 1)
.OrderBy(d => d.Nachname);
```

Im Gegensatz zu SQL können Sie die Reihenfolge von **where**- und **orderby**-Elementen vertauschen.

Nach mehreren Feldern sortieren

Allerdings gelingt es nicht, einfach mehrere **orderby**-Ausdrücke anzugeben – es wird dann einfach immer der letzte Ausdruck ausgewertet, wie in folgendem Beispiel:

```
//sortiert nur nach dem Nachnamen:
var kunden =
from d in context.Kunden
orderby d.Vorname,
orderby d.Nachname
select d;
```

Für das Sortieren nach mehreren Eigenschaften in der Abfragensyntax geben Sie die zu sortierenden Eigenschaften einfach in der gewünschten Reihenfolge an – genau wie unter SQL:

```
var kunden =
from d in context.Kunden
orderby d.Vorname, d.Nachname
select d;
```

Für eine absteigende Sortierung hängen Sie noch das Schlüsselwort **descending** an – oder, um explizit die aufsteigende Sortierung festzulegen, das Schlüsselwort **ascending**. Hier werden beide verwendet:

```
var kunden =
from d in context.Kunden
orderby d.Vorname ascending,
d.Nachname descending select d;
```

Bei der Methodensyntax gibt es das **thenby**-Schlüsselwort:

LINQ to Entities: Daten bearbeiten

In den vorangegangenen Ausgaben von DATENBANKENTWICKLER haben Sie bereits erfahren, wie Sie per ADO.NET auf die Daten der Tabellen einer Datenbank zugreifen. Nun nutzen wir nicht mehr direkt ADO.NET, sondern das Entity Framework als Datenlieferant, welches eine ganz andere Art des Zugriffs ermöglicht. Dieser Artikel zeigt, wie Sie die per Entity Data Model bereitgestellten Daten mit LINQ to Entities ändern, löschen und neu anlegen.

Beispielprojekt

Das Beispielprojekt soll über ein geeignetes Entity Data Model auf die Tabellen in einer LocalDB-Datenbank zugreifen. Wie Sie das Entity Data Model erstellen, beschreiben wir im Artikel **Entity Data Model für eine Datenbank erstellen**.

In Bild 1 sehen Sie die Klassen des Beispiels, die prinzipiell wie ein Datenmodell aussehen. Diese Klassen bilden die Grundlage der Beispiele für einige Artikel, aber später werden wir diese gegebenenfalls noch etwas erweitern. Alle Beispielmethoden finden Sie in der Klasse **Beispiele_EF_Bearbeiten.cs**. Um die Beispiele auszuprobieren, starten Sie einfach das Projekt.

Aufruf der Beispielmethoden

Damit wir die nachfolgend programmierten Methoden einfach aufrufen können, verwenden wir wieder die bereits in früheren Artikeln verwendete Klasse **Program** mit einer **Main**-Methode, die alle öffentlichen, statischen Methoden aus öffentlichen Klassen auflistet und zur Ausführung anbietet. In einer neuen Klasse namens **Beispiele_EF_Bearbeiten** legen wir dann die später vorgestellten Methoden an.

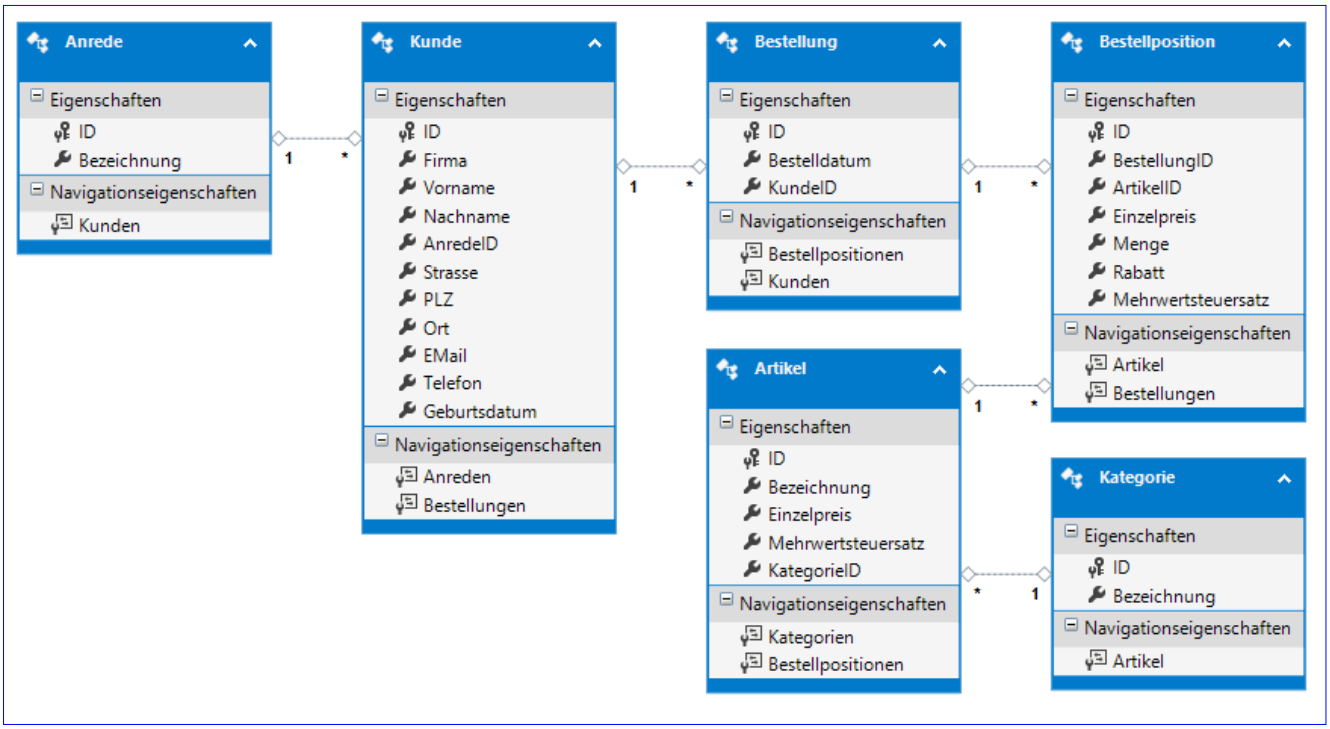


Bild 1: Klassendiagramm für die Beispiele dieses Artikels

```
public static void KundeAendern() {
    using (BestellverwaltungEntities context = new BestellverwaltungEntities()) {
        Kunde kunde = context.Kunden.Find(1);
        Console.WriteLine("Kunde vorher: {0} {1} {2}", kunde.ID, kunde.Vorname, kunde.Nachname);
        kunde.Vorname = "Andreas";
        Console.WriteLine("Kunde nachher: {0} {1} {2}", kunde.ID, kunde.Vorname, kunde.Nachname);
        context.SaveChanges();
    }
    using (BestellverwaltungEntities context = new BestellverwaltungEntities()) {
        Kunde kunde = context.Kunden.Find(1);
        Console.WriteLine("Kunde neu eingelesen: {0} {1} {2}", kunde.ID, kunde.Vorname, kunde.Nachname);
    }
    Console.ReadLine();
}
```

Listing 1: Einlesen, ändern und erneutes Einlesen eines Kunden

Einen Kunden bearbeiten

Im Artikel **LINQ to Entities: Daten abfragen** haben Sie bereits erfahren, wie Sie die Daten aus einer Tabelle in ein oder mehrere Objekte laden und diese in der Konsole ausgeben können. Genau das erledigen wir jetzt auch – mit dem Unterschied, dass wir auch noch eine Änderung an diesem Kunden vornehmen und diese Änderung speichern.

Den vollständigen Code des Beispiels sehen Sie in Listing 1. Die Methode **KundeAendern** erstellt eine Variable des Typs **BestellverwaltungEntities**, welche die verschiedenen Klassen wie **Kunde**, **Anrede** et cetera bereithält, und speichert den Verweis darauf in der Variablen **context**, welche innerhalb des **using**-Konstrukts ihre Gültigkeit behält. Über die **Find**-Methode des **Kunden**-DBSets von **context** ermittelt die Prozedur dann den Kunden mit dem Wert **1** in der Identitätsspalte, in diesem Fall im Feld **ID**. Die Methode gibt nun die Inhalte der drei Felder **ID**, **Vorname** und **Nachname** des **Kunde**-Objekts in der Konsole aus. Dann ändert sie durch eine einfache Zuweisung des Wertes **Andreas** an die Eigenschaft **Vorname** den Inhalt des Objekts. Die erneute Ausgabe zeigt den geänderten Inhalt. Nun folgt der entscheidende Schritt: Die Methode ruft die **SaveChanges**-Methode des **context**-Objekts auf. Damit sollen die Änderungen in die Datenbank übertragen werden. Um dies zu prüfen, erstellen wir einen neuen Kontext auf die Datenbank, lesen den Kundendatensatz erneut in das Objekt **kunde** ein und geben die aktuellen Werte von **ID**, **Vorname** und **Nachname** aus. Das Ergebnis: Die Änderung wurde in die Datenbank übernommen.

Wenn Sie den SQL Server-Profiler mitlaufen lassen, können Sie sich die durch die **SaveChanges** aufgerufene SQL-Anweisung ansehen (mehr dazu im Artikel **SQL Server-Interaktion mit dem Profiler verfolgen**):

```
exec sp_executesql N'UPDATE [dbo].[Kunde]
SET [Vorname] = @0
WHERE ([ID] = @1)
',N'@0 nvarchar(255),@1 int',@0=N'Andreas',@1=1
```

Hier wurde in der Tat nur das Feld **Vorname** für den Datensatz mit dem Wert **1** im Feld **ID** geändert.

Sie können die Änderungen natürlich auch direkt im SQL Server mitverfolgen, indem Sie dort die Herkunftstabelle (hier **Kunde**) mit der rechten Maustaste anklicken und den Kontextmenüeintrag **Oberste 200 Zeilen bearbeiten** auswählen. Sie können die dortige Ansicht aktualisieren, indem Sie die Tastenkombination **Strg + R** betätigen.

Einen neuen Kunden hinzufügen

Kommen wir nun zum Hinzufügen eines neuen Kunden. Hier betrachten wir zunächst, was wir bisher wissen: Wenn wir

WPF/EDM: Kundenübersicht

WPF-Fenster können zwar auch Daten ADO.NET-Datenquellen anzeigen, aber optimal sind sie für den Zugriff auf Daten aus einem Entity Data Model vorbereitet. Dieser Artikel zeigt, wie Sie die per Entity Data Model abgebildeten Daten aus einer Haupt- und einer Lookup-Tabelle in einem WPF-Fenster anzeigen. Dabei soll das Fenster die Navigation durch die Datensätze und auch das Anlegen neuer Datensätze ermöglichen. Im Vergleich zu einem früheren Artikel zeigen wir nun, wie Sie die Aufgabe manuell erledigen, also ohne die Unterstützung von Assistenten.

Um eine datenbankbasierte WPF-Anwendung zu erstellen, legen Sie zunächst ein neues Projekt auf Basis der Vorlage **Visual C#\WPF-Anwendung** an.

Dann fügen Sie ein Entity Data Model auf Basis unserer Beispieldatenbank **Bestellverwaltung** hinzu, wie es im Artikel **Entity Data Model für eine Datenbank erstellen** im Detail beschrieben wird. Hier die Kurzform ohne Screenshots und Umwege:

- Betätigen Sie die Tastenkombination **Strg + Umschalt + A** und wählen Sie im Dialog **Neues Element hinzufügen** den Eintrag **ADO.NET Entity Data Model** aus. Geben Sie als Name **BestellverwaltungModel** an und klicken Sie auf **Hinzufügen**.
- Behalten Sie im Dialog **Assistent für Entity Data Model** den Wert **EF Designer aus Datenbank** bei und klicken Sie auf **Weiter**.
- Klicken Sie unter **Wählen Sie Ihre Datenbankverbindung aus** auf **Neue Verbindung ...** und wählen Sie dort **Microsoft SQL Server** aus.
- Geben Sie im Dialog **Verbindungseigenschaften** den Servernamen ein (zum Beispiel **(localdb)\v11.0** oder **(localdb)\mssqllocaldb**, wenn Sie LocalDB verwenden).
- Wählen Sie die Datenbank aus oder geben Sie die anzuhängende Datenbankdatei an. Testen Sie die Verbindung und klicken Sie auf **OK**.
- Zurück im **Assistent für Entity Data Model** bestätigen Sie den Namen für die Verbindungseinstellungen (hier **BestellverwaltungEntities**).
- Wenn Sie möchten, bestätigen Sie die Frage, ob Sie die Datenbankdatei zum Projekt hinzufügen möchten.
- Behalten Sie **Entity Framework 6.x** als Version bei.
- Wählen Sie alle gewünschten Tabellen aus und geben Sie den Namen für den Modellnamespace an (hier **BestellverwaltungModel**).
- Klicken Sie auf **Fertigstellen**. Das Entity Data Model wird nun erstellt.

Nun nehmen Sie noch eventuell nötige Änderungen an den Namen der Entitäten oder den Namen der Entitätsmengen vor. Die Beispieldatenbank verwendet Tabellen mit Namen im Singular. Sie sollten daher die Eigenschaft **Name der Entitätenmenge** für alle Entitäten über das Modelldiagramm (hier **Bestellverwaltungmodel.edmx**) auf die Pluralform einstellen (also **Kategorien** statt **Kategorie** und so weiter). Anschließendes Speichern aktualisiert den Code. Für dieses Beispiel arbeiten wir mit den Tabellen **Kunden** und **Anreden**, wobei wir als Namen für die Entitätenmengen die Namen **Kunden** und **Anreden** übernehmen und die Entitätsklassen selbst mit **Kunde** und **Anrede** benennen.

Um die Namen der Entitätsklassen anzupassen, öffnen Sie die Datei **BestellverwaltungModel.edmx**, markieren

nacheinander die Entitäten und ändern die Bezeichnungen der Entitäten von Singular auf Plural (siehe Bild 1). Speichern nicht vergessen, um die Änderungen auf den Code zu übertragen – dies kann ein paar Sekunden dauern.

Kunden im DataGrid

Als Erstes wollen wir die Kunden der Tabelle **Kunden** im DataGrid anbieten. Dieses Datagrid legen wir in einem neuen Fenster namens **KundenuebersichtEinfach** an, das Sie per **Strg + Umschalt + A** und Auswahl des Eintrags Fenster sowie Angabe der Bezeichnung **KundenuebersichtEinfach** anlegen.

Um dieses Fenster vom Fenster **MainWindow.xaml** aus zu öffnen, fügen wir diesem eine neue Schaltfläche namens **btnKundenuebersichtEinfach** hinzu:

```
<Button x:Name="btnKundenuebersichtEinf
ach" Content="Kundenübersicht einfach"
HorizontalAlignment="Left" Margin="31,20,0,0"
VerticalAlignment="Top" Width="143" Click="btnKundenuebers
ichtEinfach_Click"/>
```

Für die Eigenschaft **Click** hinterlegen wir eine Methode, die wie folgt aussieht:

```
private void btnKundenuebersichtEinfach_Click(
                    object sender, RoutedEventArgs e) {
    KundenuebersichtEinfach wnd =
                        new KundenuebersichtEinfach();
    wnd.Show();
}
```

Kunden anzeigen – einfache Variante

Dem Fenster **Kundenuebersicht-Einfach** fügen Sie nun das **Datagrid**-Steuerelement hinzu, welches die Felder der Tabelle **Kunden** anzeigen soll. Der XAML-Code sieht dann etwa wie folgt aus:

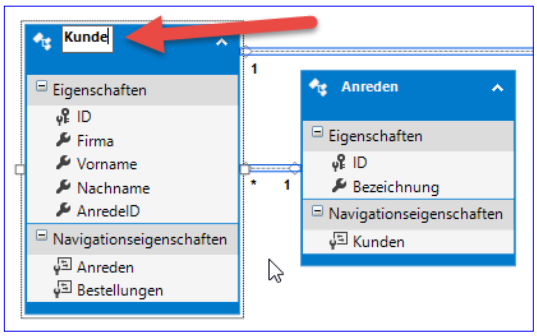


Bild 1: Ändern der Entitätsnamen

```
<Window x:Class="Bestellverwaltung_EDM.Kundenuebersicht"
...
        Title="Kundenuebersicht" Height="300" Width="600">
    <Grid>
        <DataGrid x:Name="dgKunden"
            HorizontalAlignment="Left" Margin="10,10,10,10"
            VerticalAlignment="Top" />
    </Grid>
</Window>
```

Sie sehen hier keinerlei Datenbindungseigenschaften. Dies erledigen wir direkt vom Code aus. Der komplette Code der Klasse sieht so aus:

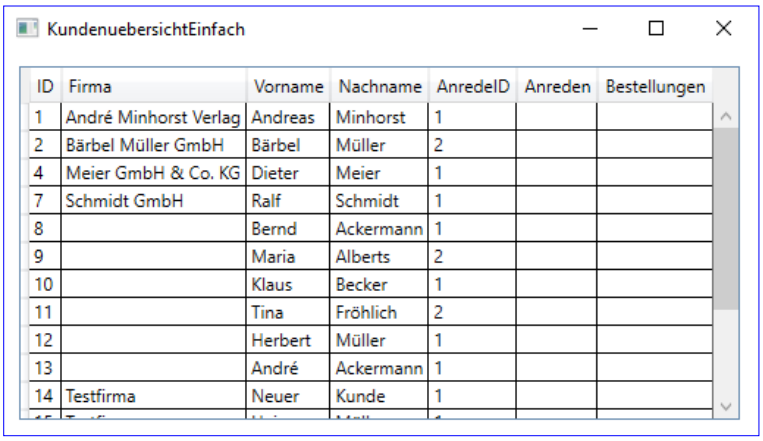
```
using System.Linq;
using System.Windows;

namespace Bestellverwaltung_EDM {
    public partial class KundenuebersichtEinfach : Window {
        public KundenuebersichtEinfach() {
            InitializeComponent();
            using (BestellverwaltungEntities DBContext =
                        new BestellverwaltungEntities()) {
                var kunden = DBContext.Kunden.ToList();
                dgKunden.ItemsSource = kunden;
            };
        }
    }
}
```

Hier verwenden wir die im Artikel **LINQ to Entities: Daten abfragen** ausführlich erläuterten Techniken, um die Daten aus einer Tabelle über das Entity Data Model in eine Variable zu füllen. Dazu erstellen wir einen Datenbank-Kontext des Typs **BestellverwaltungEntities**. Dieses bietet die Entitätsauflistung **Kunden**, die wir mit der **ToList**-Methode in eine Liste füllen und an die Variable

kunden übergeben. Diese landet schließlich über die Eigenschaft **ItemsSource** im DataGrid:

```
using (BestellverwaltungEntities DBContext =
            new BestellverwaltungEntities()) {
    var kunden = DBContext.Kunden.ToList();
    dgKunden.ItemsSource = kunden;
};
```

Das Ergebnis sehen Sie in Bild 2. Weil das Attribut **AutoGenerateColumns** in der XAML-Definition des **DataGrid**-Steuerelements nicht explizit gesetzt wurde, hat dieses den Standardwert **True**, was dazu führt, dass alle Felder der Datenquelle im **DataGrid**-Steuerelement abgebildet werden. Da wir die Daten mit der Methode **ToList()** in die Variable **kunden** geschrieben haben, werden die verknüpften Objekte **Anreden** und **Bestellungen** nicht gefüllt. Damit können Sie nun bereits die üblichen Aktionen wie etwa das Sortieren durchführen, aber es werden noch nicht einmal die Anreden angezeigt, sondern nur die Werte des Feldes **AnredeID** – und die beiden verknüpften Eigenschaften **Anreden** und **Bestellungen** wollen wir auch loswerden.



ID	Firma	Vorname	Nachname	AnredeID	Anreden	Bestellungen
1	André Minhorst Verlag	Andreas	Minhorst	1		
2	Bärbel Müller GmbH	Bärbel	Müller	2		
4	Meier GmbH & Co. KG	Dieter	Meier	1		
7	Schmidt GmbH	Ralf	Schmidt	1		
8		Bernd	Ackermann	1		
9		Maria	Alberts	2		
10		Klaus	Becker	1		
11		Tina	Fröhlich	2		
12		Herbert	Müller	1		
13		André	Ackermann	1		
14	Testfirma	Neuer	Kunde	1		

Bild 2: Erster Entwurf des **DataGrid**-Steuerelements mit Kundendaten

Manuelle Definition der anzuzeigenden Felder

Wie wir oben beschrieben haben, werden die Felder aktuell angezeigt, weil die Option **AutoGenerateColumns** für das **DataGrid**-Steuerelement aktiviert ist. Wir wollen aber individuelle Einstellungen für die angezeigten Daten vornehmen. Daher deaktivieren wir zunächst mit dem Attribut **AutoGenerateColumns="False"** die automatische Generierung. Außerdem definieren wir selbst die anzuzeigenden Spalten und deren Spaltenüberschriften und Inhalte. Dies sieht dann wie in der Definition aus Listing 1 aus, wo wir für Spalten jeweils ein Element des Typs **DataGridTextColumn**

```
<Window x:Class="Bestellverwaltung_EDM.Kundenuebersicht_Columns" ...
                                                    Title="Kundenuebersicht_Columns" Height="300" Width="600">
    <Grid>
        <DataGrid x:Name="dgKunden" HorizontalAlignment="Left" Margin="10,10,10,10" VerticalAlignment="Top"
                                                                        AutoGenerateColumns="False">
            <DataGrid.Columns>
                <DataGridTextColumn Header="ID" Binding="{Binding ID}" Width="30"></DataGridTextColumn>
                <DataGridTextColumn Header="Firma" Binding="{Binding Firma}" Width="*"></DataGridTextColumn>
                <DataGridTextColumn Header="Anrede" Binding="{Binding Anrede}" Width="50"></DataGridTextColumn>
                <DataGridTextColumn Header="Vorname" Binding="{Binding Vorname}" Width="*"></DataGridTextColumn>
                <DataGridTextColumn Header="Nachname" Binding="{Binding Nachname}" Width="*"></DataGridTextColumn>
                <DataGridTextColumn Header="Straße" Binding="{Binding Strasse}" Width="*"></DataGridTextColumn>
                <DataGridTextColumn Header="PLZ" Binding="{Binding PLZ}" Width="*"></DataGridTextColumn>
                <DataGridTextColumn Header="Ort" Binding="{Binding Ort}" Width="*"></DataGridTextColumn>
            </DataGrid.Columns>
        </DataGrid>
    </Grid>
</Window>
```

Listing 1: Benutzerdefinierte Festlegung der Spalten

verwenden. Als Datenbindung geben wir für die Eigenschaft **Binding** jeweils Werte wie **{Binding ID}** an.

Den Code dieses Beispiels finden Sie im Fenster **Kundenuebersicht_Columns.xaml**. Die Ansicht entspricht weitgehend der aus dem ersten Beispiel.

Kunden per Eigenschaft füllen

Nun gehen wir einen Schritt weiter: Bisher haben wir die Datenquelle im C#-Code im Code behind-Modul für die Eigenschaft ItemsSource angegeben. Nun wollen wir die Zuweisung in den XAML-Code verschieben. All dies erledigen wir in dem Beispielfenster **KundenuebersichtBinding.xaml**. Warum wollen wir die folgende Zeile gern aus dem Code behind-Modul heraushaben?

```
dgKunden.ItemsSource = kunden;
```

Weil wir so die Benutzeroberfläche und die Anwendungslogik besser voneinander trennen können. Später, also in folgenden Ausgaben dieses Magazins, werden wir uns mit MVVM beschäftigen, einem Entwurfsmuster, mit dem genau diese Trennung erreicht werden soll. Ein erster Schritt dorthin ist, dass wir die Eigenschaften und Inhalte der Benutzeroberfläche nicht direkt per Code ändern, sondern dass wir per Code Eigenschaften bereitstellen, welche die Elemente der Benutzeroberfläche dann referenzieren. In diesem Fall wollen wir also nicht vom Code aus die Datenquelle für das **DataGrid**-Steuerelement festlegen, sondern nur eine Eigenschaft bereitstellen, welche die Daten liefert (in diesem Fall eine Liste von **Kunde**-Objekten). Auf diese Eigenschaft greifen wir dann mit einem entsprechenden Attribut vom XAML-Code des Fensters zu.

Dies sieht dann wie in Listing 2 aus. Hier sind zwei verschiedene Eigenschaften wichtig:

- Die Eigenschaft **ItemsSource** des **DataGrid**-Steuerelements legt fest, von welcher Eigenschaft die Daten bezogen werden.

- Die Eigenschaft **Binding** legt fest, welche Eigenschaft/welches Feld der mit **ItemsSource** definierten Datenquelle als Inhalt der Textspalten des **DataGrid**-Steuerelements genutzt werden.

Woher aber weiß die XAML-Definition, woher **Kunden** stammt? Dies legen wir in der Code behind-Datei fest. Diese verwendet folgende Namespaces, wobei **System.Collections.Generic** neu ist:

```
using System.Collections.Generic;
using System.Linq;
using System.Windows;
```

```
<Window x:Class="Bestellverwaltung_EDM.KundenuebersichtBinding" ...
        Title="KundenuebersichtBinding" Height="300" Width="300">
    <Grid>
        <DataGrid x:Name="dgKunden" HorizontalAlignment="Left" Margin="10,10,10,10" VerticalAlignment="Top"
                                                    ItemsSource="{Binding Kunden}" AutoGenerateColumns="False">
            <DataGrid.Columns>
                <DataGridTextColumn Header="ID" Binding="{Binding ID}" Width="30"></DataGridTextColumn>
                <DataGridTextColumn Header="Firma" Binding="{Binding Firma}" Width="*"></DataGridTextColumn>
            ...
            </DataGrid.Columns>
        </DataGrid>
    </Grid>
</Window>
```

Listing 2: Die Datenbindung erfolgt nun komplett über Eigenschaften.

WPF/EDM: Kundendetails

Wenn Sie aus einer Kundenübersicht die Details eines Kunden anzeigen oder einen neuen Kunden anlegen möchten, benötigen Sie ein geeignetes weiteres Fenster. Dieses soll die Daten des zu bearbeitenden Kundendatensatzes oder auch einen neuen, leeren Datensatz anzeigen – je nach Anforderung. Wie Sie dies auf Basis von Daten aus einem Entity Data Model erledigen, zeigt dieser Artikel.

Voraussetzungen

Für das Nachvollziehen dieses Beispiels erstellen Sie ein Entity Data Model wie im Artikel **WPF/EDM: Kundenübersicht** beschrieben. Das dort beschriebene Fenster mit einer Übersicht der gespeicherten Kunden verwenden wir, um das im vorliegenden Artikel beschriebene Detailfenster zu öffnen.

Fenster zum Hinzufügen und Bearbeiten von Kunden

Bevor wir Kunden bearbeiten oder hinzufügen können, benötigen wir ein entsprechendes Fenster. Dieses definieren wir mit der **.xaml**-Datei aus Listing 1 (gekürzte Fassung).

Hier haben wir ein Raster von vier Spalten und sieben Zeilen, wobei die oberen Zeilen für die Beschriftungen und die Felder des Datensatzes und die unteren für Schaltflächen zum Speichern und zum Abbrechen der aktuellen Änderungen vorgesehen sind. Der Entwurf sieht wie in Bild 1 aus. Die Bindung der Felder verläuft wie schon beim Übersichtsfenster, nur dass hier kein DataGrid mit einzelnen Spalten, sondern jeweils unabhängige Textfelder und ein Kombinationsfeld zum Einsatz kommen. Die Bindung erfolgt über die entsprechenden Eigenschaften des **Kunde**-Objekts.

Aus Gründen, die wir später erläutern, nutzen wir hier jedoch eine ungebundene Kopie des zu bearbeitenden **Kunde**-Objekts beziehungsweise ein neues **Kunde**-Objekt, wenn ein neuer Kunde angelegt werden soll. Dieses Objekt nennen wir **KundeTemp**. Da wir in der Definition kein übergeordnetes Element haben, dem wir das **KundeTemp**-Objekt zuweisen

```
<Window x:Class="Bestellverwaltung_EDM.Kundendetails" ...Title="Kundendetails" Height="300" Width="300" Icon="users3.ico">
    <Grid>
        //... Definition von vier Spalten und sieben Zeilen
        <TextBlock Grid.Column="0" Grid.Row="0" Margin="3" Text="ID:" />
        <TextBlock Grid.Column="0" Grid.Row="1" Margin="3" Text="Firma:" />
        <TextBlock Grid.Column="0" Grid.Row="2" Margin="3" Text="Anrede:" />
        <TextBlock Grid.Column="0" Grid.Row="3" Margin="3" Text="Vorname:" />
        <TextBlock Grid.Column="0" Grid.Row="4" Margin="3" Text="Nachname:" />
        <TextBox ... Text="{Binding KundeTemp.ID}" IsEnabled="False"/>
        <TextBox ... Text="{Binding KundeTemp.Firma}" />
        <ComboBox ... ItemsSource="{Binding Anreden}" SelectedItem="{Binding KundeTemp.Anreden}"
                                                DisplayMemberPath="Bezeichnung" SelectedValuePath="Id" />
        <TextBox ... Text="{Binding KundeTemp.Vorname}" />
        <TextBox ... Text="{Binding KundeTemp.Nachname}" />
        <Button x:Name="btnAbbrechen" ... Content="Abbrechen" Click="btnAbbrechen_Click" />
        <Button x:Name="btnSpeichern" ... Content="Speichern" Click="btnSpeichern_Click" />
    </Grid>
</Window>
```

Listing 1: Benutzerdefinierte Festlegung der Spalten

können (etwa der **ItemsSource** eines **DataGrid**-Elements), tragen wir die Werte für die Eigenschaft **Text** der **Textbox**-Steuerelemente in der Form **{Binding KundeTemp.<Eigenschaftsname>}** ein.

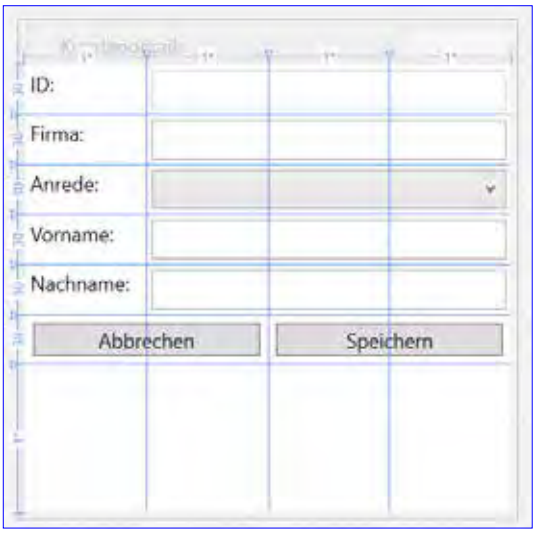


Bild 1: Entwurf des Detailfensters

Kombinationsfeld für die Anreden

Das Kombinationsfeld zur Anzeige der Anreden füllen wir mit mehr als einer Eigenschaft. Das Attribut **ItemsSource** erhält den Wert **{Binding Anreden}**, wodurch die Eigenschaft **Anreden** der Code behind-Klasse als Datenherkunft dieses Steuerelements dient.

Damit es den für diesen Kunden festgelegten Eintrag anzeigt, legen wir für das Attribut **SelectedItem** den Wert **{Binding KundeTemp.Anreden}** fest. Dies liefert genau das benötigte **Anrede**-Objekt.

Das Attribut **DisplayMemberPath** gibt an, welche Eigenschaft des enthaltenen Objekts angezeigt werden soll, hier also die Eigenschaft **Bezeichnung**. **SelectedValuePath** erhält die Eigenschaft **ID** des gewählten **Anrede**-Objekts.

Öffnen des jeweiligen Formulars übergeben konnten, beispielsweise mit

```
DoCmd.OpenForm "frmKunde",
DataMode:=acFormAdd
```

für einen neuen Datensatz oder

```
DoCmd.OpenForm "frmKun-
de", DataMode:=acFormEdit,
WhereCondition:="ID = " & KundeID
```

zum Bearbeiten eines vorhandenen Datensatzes. Unter C# können wir solche Parameter frei definieren, und zwar durch Anpassung der Konstruktor-Methode. Bevor wir dazu kommen, ein kurzer Blick auf die Variablen dieser Klasse:

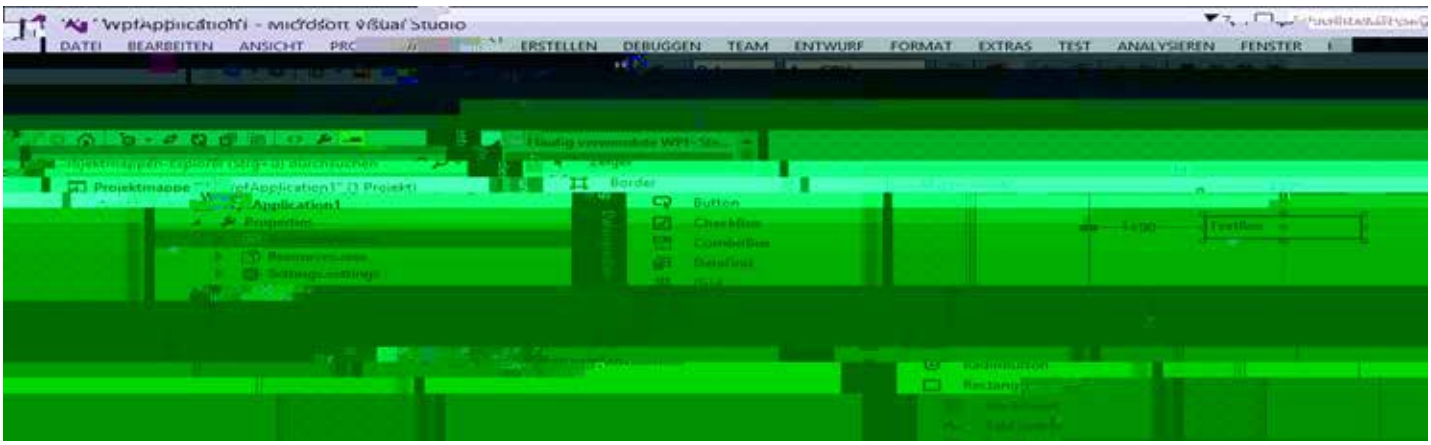
```
public Kunde KundeTemp { get; set; }
private Kunde KundeAktuell;
BestellverwaltungEntities DBContext;
private List<Anreden> anreden;
```

KundeTemp nimmt ein temporäres **Kunde**-Objekt auf, wobei es sich wahlweise um den zu editierenden oder den neu zu erstellenden Kunden handelt.

AUSGABE 05/2016

DATENBANK ENTWICKLER

MAGAZIN FÜR DIE DATENBANKENTWICKLUNG MIT VISUAL STUDIO FÜR DESKTOP, WEB UND CO.



TOP-THEMEN:

SQL SERVER	SQL Server: Datenbanken per Skript kopieren	SEITE 3
C#-GRUNDLAGEN	Die MessageBox-Klasse	SEITE 11
WPF-BASICS	Ribbons mit WPF	SEITE 21
WPF-BASICS	EDM: Einfaches Detailfenster	SEITE 34
WPF-BASICS	EDM: Lookup-Kombinationsfelder	SEITE 47

SQL Server: Datenbanken per Skript kopieren

Die Beispieldatenbanken auf Basis des SQL Servers sollen Sie schnell und unkompliziert auf Ihrem System nutzbar machen können. Leider geht das nicht immer so einfach – zum Beispiel, weil verschiedene Versionen des SQL Servers uns einen Strich durch die Rechnung machen. Also stellen wir in diesem Artikel eine einfache Lösung vor, wie Sie die Beispieldatenbanken nach dem Download schnell nutzen können – nämlich indem wir diese als Skript bereitstellen, das die komplette Definition der Datenbank sowie die enthaltenen Daten anlegt.

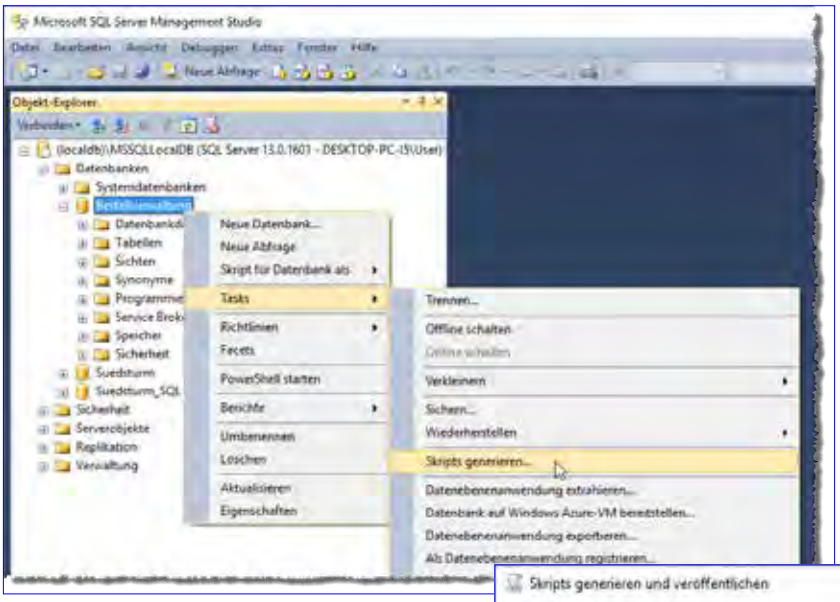


Bild 1: Hinzufügen der Verbindung

Für die Nutzung der Beispieldatenbanken verwenden Sie optimalerweise neben Ihrer SQL Server-Installation, bei der es sich auch um die einfache LocalDB-Variante handeln kann, noch das kostenlos verfügbar SQL Server Management Studio.

Wenn ich die Beispieldatenbank für den Download vorbereite, habe ich diese bereits im SQL Server Management Studio eingebunden – in diesem Beispiel in der Version 2014. Hier kann ich dann im Objekt-Explorer unter **Datenbanken** die gewünschte Datenbank auswählen (hier **Bestellverwaltung**), das Kontextmenü anzeigen und dort den Befehl **Tasks|Skripts generieren...** ausführen (siehe Bild 1).

Skript-Assistent starten

Dies öffnet den Dialog aus Bild 2. Hier können Sie entweder die Option **Skripterstellung für gesamte Datenbank und alle Datenbankobjekte** beibehalten oder auf **Bestimmte Datenbankobjekte auswählen** wechseln und hier eine individuelle Auswahl festlegen. Bei unserer begrenzten Menge von Datenbankobjekten sind beide

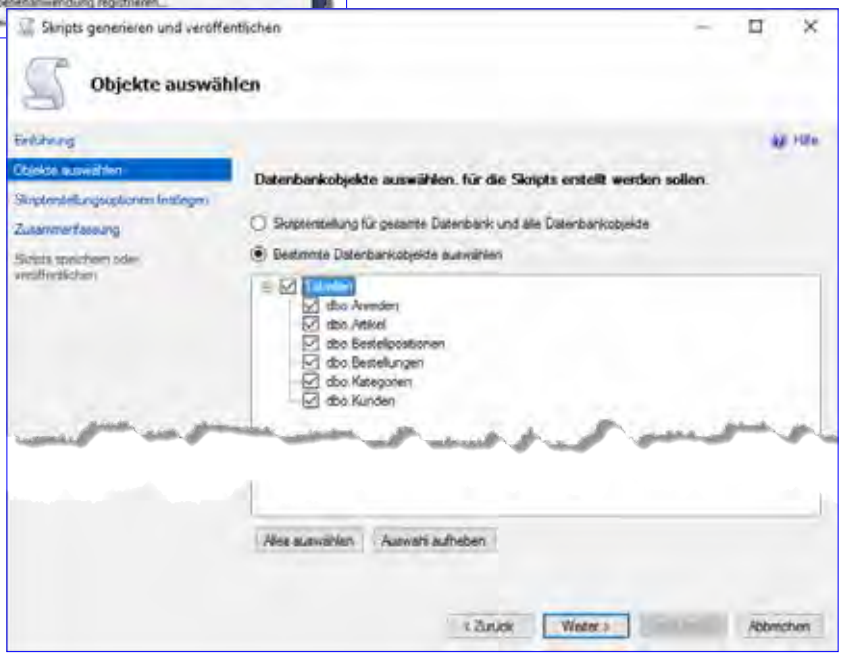


Bild 2: Auswahl der zu exportierenden Datenbankobjekte

Methoden gleich schnell ausgeführt. Aber es gibt einen entscheidenden Unterschied:

Wenn Sie **Bestimmte Datenbankobjekte auswählen** selektieren, werden, selbst wenn Sie dann alle Elemente selektieren, nur die **CREATE TABLE**-Statements für die einzelnen Tabellen (beziehungsweise **CREATE...**-Anweisungen für die übrigen Objekttypen) erstellt, aber kein **CREATE DATABASE** – Sie können mit dem erstellten Skript dann keine neue Datenbank generieren, sondern lediglich die Datenbankobjekte in eine bestehende Datenbank eintragen.

Wenn Sie eine komplett neue Datenbank erstellen wollen, wählen Sie die Option **Skripterstellung für gesamte Datenbank und alle Datenbankobjekte**.

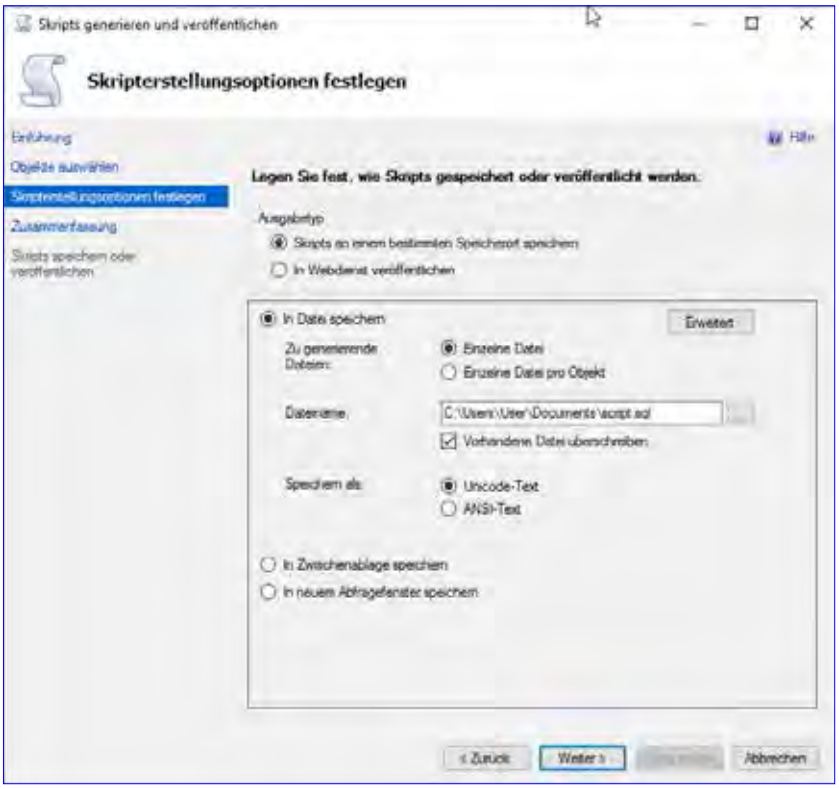


Bild 3: Festlegen der Optionen zum Exportieren der Datenbankobjekte

Einen Dialogschritt weiter unter **Skripterstellungsoptionen festlegen** finden Sie die Optionen aus Bild 3. Hier legen Sie zunächst als Ausgabetyp die Option **Skripts an einem bestimmten Speicherort speichern** aus. Darunter finden Sie reichlich Möglichkeiten, um das Ziel des Exports zu definieren.

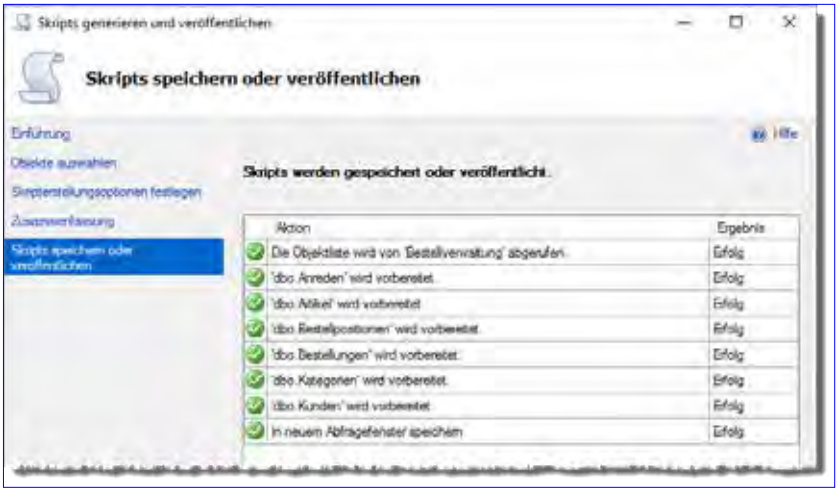


Bild 4: Ergebnis des Exports

Die erste Option geht davon aus, dass Sie die Definition aller Elemente in einer einzelnen Datei speichern wollen. Dazu geben Sie dann den gewünschten Dateinamen an und ob eventuell bereits vorhandene Dateien gleichen Namens überschrieben werden sollen. Außerdem stellen Sie hier das Exportformat fest und wählen dabei zwischen **Unicode-Text** und **ANSI-Text**.

Vielleicht wollen Sie aber gar keine Datei anlegen, sondern sich das Skript erst einmal ansehen oder es in die Zwischenablage speichern? Beides ist kein Problem, wenn Sie die zweite Option **In Zwischenablage speichern** oder die dritte Option **In neuem Abfragefenster speichern** selektieren.

Wir wählen letztere Version, um uns das Ergebnis zunächst anzusehen. Der Assistent liefert uns zur Unterstützung einen

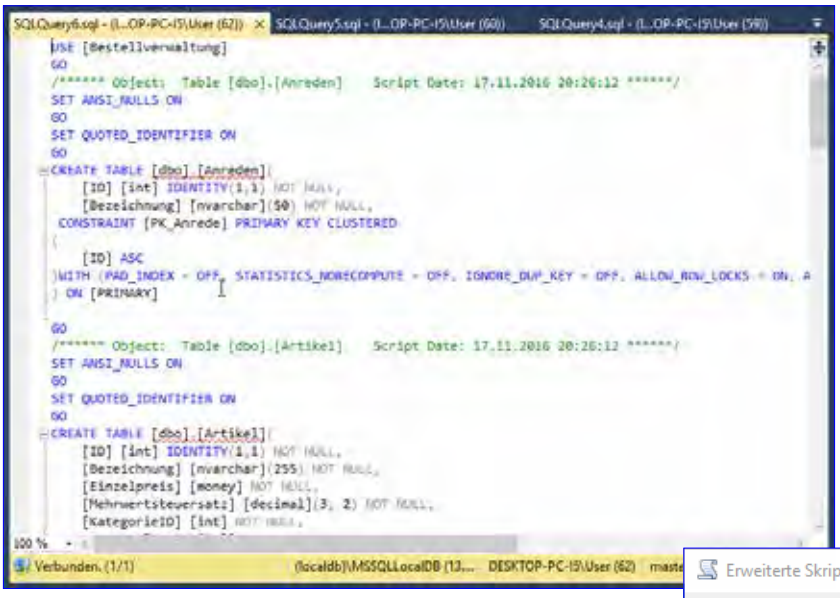


Bild 5: Das fertige SQL-Skript

Bericht mit den Ergebnissen der Skripterstellung (siehe Bild 4).

Ergebnis ansehen

Nach dem Schließen des Assistenten können Sie zum SQL Server Management Studio zurückwechseln und finden dort ein neues Abfragefenster wie in Bild 5 vor. Das sieht schon recht gut aus – das Skript liefert die SQL-Anweisungen zum Erstellen der Tabellen unserer Datenbank. Wenn wir jedoch ein wenig weiter nach unten scrollen, stellen wir fest, dass dies leider alles ist: Neben ein paar weiteren Anweisungen zum Ändern einiger Felder in Fremdschlüsselfelder folgen keine weiteren Befehle. Damit erstellt das Skript zwar die gewünschte Tabellenstruktur, füllt die Tabellen aber nicht mit den Daten der in das Skript exportierten Datenbank. Außerdem fällt auf, dass man auch die Datenbank noch selbst anlegen muss – die erste Anweisung des Skripts lautet **USE [Bestellverwaltung]** statt **CREATE DATABASE [Bestellverwaltung]**. Eine bestehende Datenbank dieses Namens ist also Voraussetzung für das Funktionieren des Skripts.

Anpassen der »Weiteren Einstellungen«

Doch wir haben noch nicht alle Möglichkeiten des Skript-Assistenten genutzt, denn wir haben die Schaltfläche **Erweitert** im Schritt **Skripterstellungsoptionen festlegen** ignoriert.

Wenn wir den Vorgang nochmal starten und in diesem Schritt die erweiterten Optionen anzeigen lassen, erhalten wir den Dialog aus Bild 6. Hier finden Sie nun eine ganze Reihe Optionen, mit denen Sie das Aussehen des Exports sehr fein einstellen können. Für uns sind hier die folgenden Optionen interessant:

Datentypen, für die ein Skript erstellt wird: Legt fest, ob Sie nur die Daten, nur das Schema oder Schema und Daten

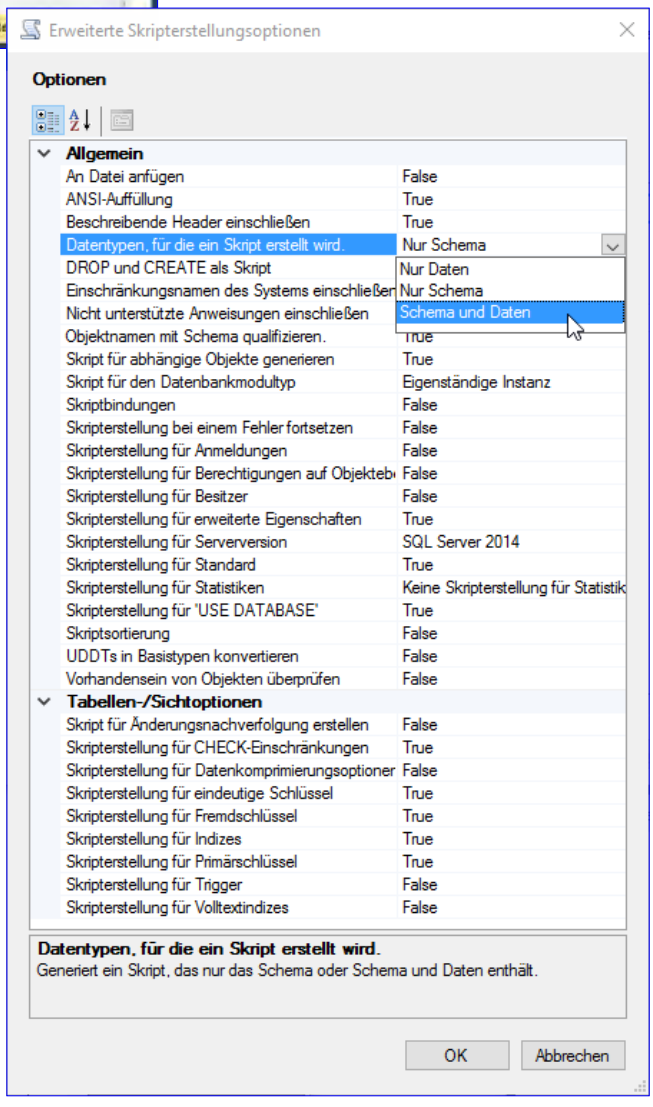


Bild 6: Optionen für das Erstellen des Skripts

exportieren wollen. Wir benötigen in diesem Fall Schema und Daten.

DROP und CREATE als Skript: Dies legt fest, ob zu jedem zu berücksichtigenden Datenbankobjekt eine **DROP**-Anweisung (zum Löschen), eine **CREATE**-Anweisung (zum Erstellen) oder beide angelegt werden sollen. Wir benötigen auf jeden Fall die **CREATE**-Anweisungen. Wenn Sie die Datenbankobjekte in eine bestehende Datenbank importieren wollen, die bereits Datenbankobjekte gleichen Namens enthält, sollten Sie zusätzlich die **DROP**-Anweisungen hinzufügen. Sie können auch ein Skript erstellen, das lediglich Datenbankobjekte löscht!

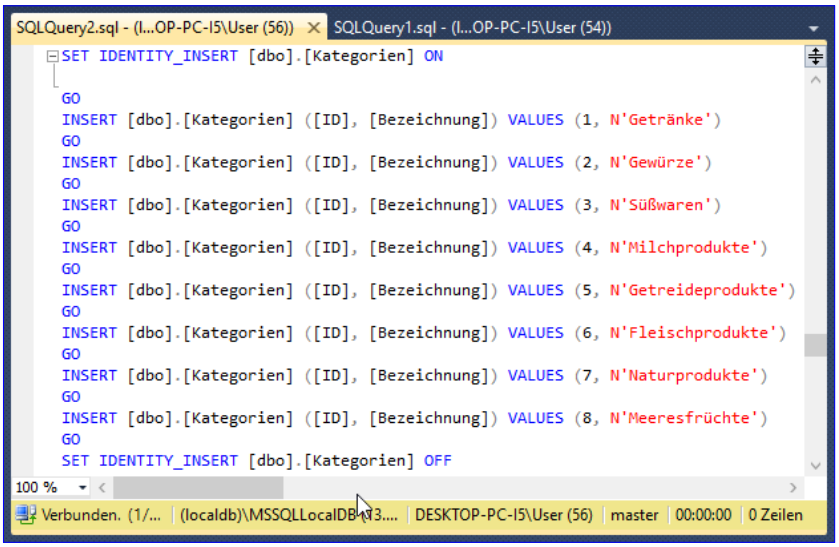


Bild 7: Skript mit **INSERT INTO**-Anweisungen

Mit **Skripterstellung für Serverversion** können Sie eine Zielversion angeben. Hier sind sie aus verständlichen Gründen auf alle zum Zeitpunkt der Veröffentlichung dieser Version von SQL Server Management Studio verfügbaren Versionen des SQL Servers beschränkt.

Wenn Sie die Skriptgenerierung nun erneut starten, erhalten Sie im erstellten Skript auch die SQL-Anweisungen, welche die Daten der Ursprungsdatenbank in die neu erstellten Tabellen einträgt (siehe Bild 7).

Datenbank neu erstellen

Auf die oben beschriebene Weise werden nun unter anderem die Beispielbanken für das Magazin bereitgestellt. Sie können die resultierenden Skriptdateien nun dazu nutzen, die Beispieldatenbank auf Ihrem eigenen Rechner anzulegen. Dazu müssen Sie lediglich eine der folgenden Aktionen durchführen:

- Das Skript etwa aus einem Texteditor in die Zwischenablage kopieren und von dort in einer neuen Abfrage im SQL Server Management Studio einfügen. Nach dem Betätigen der Taste **F5** wird das Skript nun ausgeführt und die Datenbank samt Tabellen und Daten erstellt.
- Sie können auch eine Datei mit dem Skript, beispielsweise **Bestellverwaltung.sql**, mit dem SQL Server Management Studio öffnen und dann per **F5** ausführen.

In beiden Fällen kann es natürlich zu Problemen kommen, die entweder mit der Kompatibilität des Skriptes mit der verwendeten Version oder auch mit eventuell bereits vorhandenen Datenbankdateien zusammenhängen. Dann werden die fehlerhaften Zeilen jedoch ausgegeben – beim Versuch, eine vorhandene Datenbank zu überschreiben, etwa so:

```
Meldung 1801, Ebene 16, Status 3, Zeile 4
Database 'Bestellverwaltung' already exists. Choose a
different database name.
Meldung 5069, Ebene 16, Status 1, Zeile 46
ALTER DATABASE statement failed.
Die Abfrage wurde vom Benutzer abgebrochen.
```

Datenbank in Visual Studio erstellen

Wenn Sie SQL Server Management Studio nicht installiert haben oder lieber alles von einer Plattform aus erledigen, können Sie die neue SQL Server-Datenbank auch von Visual Studio aus installieren. Dazu wählen Sie den Menübefehl **Ansicht|SQL Server-Objekt-Explorer** aus, was den gleichnamigen Bereich in Visual Studio einblendet. Dort können Sie über den Kontextmenü-Eintrag **Neue Abfrage...** des Daten-

Die MessageBox-Klasse

Die MessageBox-Klasse unter C# bietet ähnliche Features wie die von VBA bekannte MsgBox-Funktion. Allerdings ist es erstens eine Klasse und zweitens kann sie natürlich doch noch einiges mehr als das VBA-Pendant. Dieser Artikel zeigt, wie Sie Meldungsfenster unter C# anzeigen und diese mit Texten, Schaltflächen und anderen Elementen ausstatten. Schließlich interessiert uns auch noch, wie wir die vom Benutzer angeklickten Schaltflächen und somit die Antworten auf die in der Meldung gestellten Fragen auswerten können.

Unter C# nutzen wir also eine Klasse namens **MessageBox**. Diese ist eine statische Klasse, das heißt, dass Sie diese nicht instanzieren müssen, bevor Sie auf ihre Methoden und Eigenschaften zugreifen. Während Sie unter VBA einfach die **MsgBox**-Funktion unter Angabe des anzuzeigenden Textes aufgerufen haben, brauchen Sie bei der **MessageBox** eine entsprechende Methode, die in diesem Fall **Show** heißt.

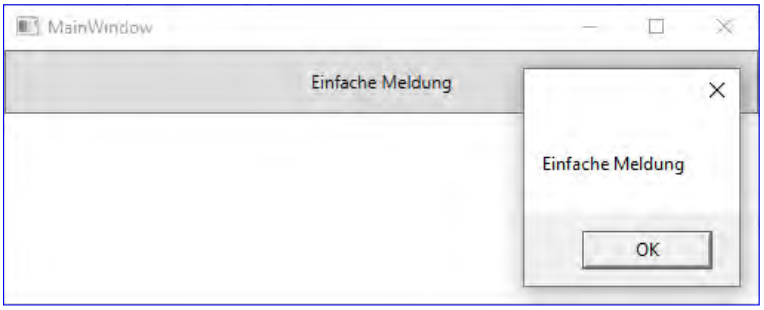


Bild 1: MessageBox mit WPF

Im Beispielprojekt zu diesem Artikel haben wir ein einfaches WPF-Fenster mit Schaltflächen gefüllt, welche die in diesem Artikel vorgestellten Meldungsfenster aufruft (siehe Bild 1). Das erste soll einfach nur einen Text anzeigen. Die Methode, die durch das **Click**-Ereignis der entsprechenden Schaltfläche aufgerufen wird, sieht wie folgt aus und nutzt die erste der zwölf angebotenen Überladungen:

```
private void btnEinfacheMeldung_Click(object sender,
        RoutedEventArgs e) {
    MessageBox.Show("Einfache Meldung");
}
```

Im Gegensatz etwa zum **Window**-Objekt, das die beiden Methoden **Show** und **ShowDialog** zum Anzeigen anbietet, gibt die **MessageBox**-Klasse nur die **Show**-Methode her. Das bedeutet allerdings nicht, dass Meldungsfenster nicht als modale Fenster ausgegeben werden – ganz im Gegenteil: **Show** liefert immer ein modales Fenster. Dementsprechend müssen Sie es erst mit einer der angezeigten Schaltflächen schließen, bevor die aufrufende Methode fortgesetzt wird.

Titel anzeigen

Mit der folgenden Überladung übergeben Sie im zweiten Parameter den anzuzeigenden Titel für die **MessageBox**:

```
MessageBox.Show("Meldung mit Titel", "Titelzeile");
```

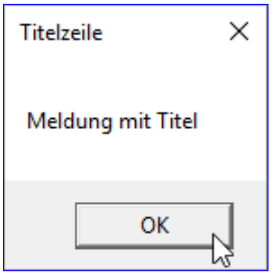


Bild 2: MessageBox mit Titelzeile

Dies sieht dann etwa wie in Bild 2 aus.

Schaltflächen der MessageBox

Die **MessageBox**-Klasse kann natürlich noch mehr Schaltflächen liefern als nur eine **OK**-Schaltfläche. Dazu verwenden Sie eine weitere Überladung, die als ersten Parameter den Text, als zweiten den Titel und als dritten eine Konstante für die anzuzeigenden Schalt-

flächen entgegennimmt. Die möglichen Werte lauten hier (siehe auch Bild 3):

- **OK**
- **OKCancel**
- **YesNo**
- **YesNoCancel**

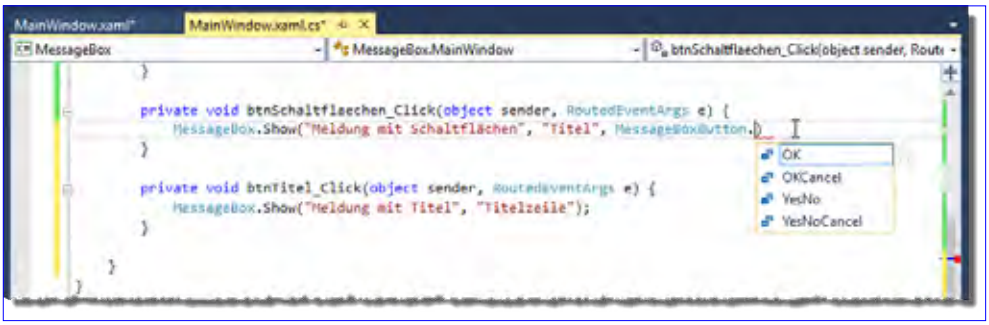


Bild 3: Auswahl der Schaltflächen-Kombinationen

Die folgende Anweisung ruft beispielsweise das Meldungsfenster aus Bild 4 hervor:

```
MessageBox.Show("Meldung mit Schaltflächen", "Titel",
MessageBoxButton.YesNoCancel);
```

Hier ist noch anzumerken, dass die **Schließen**-Schaltfläche oben rechts nur aktiviert ist, wenn entweder nur eine Schaltfläche angezeigt wird (also **OK**) oder wenn mehrere Schaltflächen aktiviert sind und sich die Schaltfläche **Cancel** darunter befindet.

Auf Schaltflächen reagieren

Wenn Sie nicht nur eine einfache **OK**-Schaltfläche nutzen, wollen Sie sicher erfahren, mit welcher Schaltfläche der Benutzer die **MessageBox** geschlossen hat. In diesem Fall weisen Sie das Ergebnis einfach einer Variablen zu und werten den Inhalt dieser Variablen danach aus. Die Variable erhält den Typ **MessageBoxResult** und wird wie folgt zugewiesen:

```
MessageBoxResult result = MessageBox.Show("Klicken Sie auf
eine der Schaltflächen.", "Ja oder Nein", MessageBoxBut-
ton.YesNo);
```

Dies liefert dann die MessageBox aus Bild 5. Danach können Sie den Wert per **if**-Bedingung prüfen und eine entsprechende Bestätigung ausgeben – natürlich per **MessageBox**:

```
if (result==MessageBoxResult.Yes) {
    MessageBox.Show("Sie haben auf 'Ja' geklickt.");
}
else {
    MessageBox.Show("Sie haben auf 'Nein' geklickt.");
}
```

Für die Antwort gibt es die folgenden Konstanten:

- **Cancel**
- **No**
- **Ok**
- **Yes**

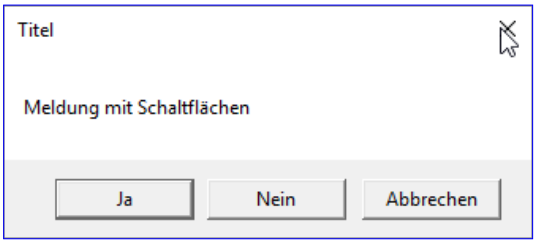


Bild 4: MessageBox mit drei Schaltflächen

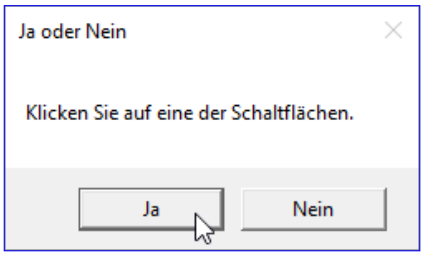


Bild 5: Testen der beiden Schaltflächen

Wie oben erwähnt, wird die **Schließen**-Schaltfläche des Fensters nur aktiviert, wenn entweder nur die **OK**-Schaltfläche oder aber eine der Kombinationen mit der **Cancel**-

InputBox im Eigenbau

Wer von Access/VBA kommt, kann unter WPF die »MsgBox«-Anweisung leicht mit den Funktionen der Klasse »MessageBox« ersetzen. Allerdings sucht man vergeblich nach einem Pendant zur InputBox-Funktion, die ja ein einfaches Eingabefenster für die Eingabe eines Textes zur Verfügung stellt. Doch das ist kein Problem: Unter C#/WPF können Sie ja eigene Fenster erstellen. Warum dann nicht ein Fenster mit Titel, Text, Eingabefeld und OK/Abbrechen-Schaltflächen bauen und dieses bei Bedarf aufrufen?

Vorgaben InputBox

Was soll eine InputBox überhaupt alles leisten? Wir richten uns im Wesentlichen nach der InputBox von Access/VBA. Diese wird wie folgt aufgerufen, wenn man die hinteren Parameter etwa für die Position und für die Onlinehilfe weglässt:

```
Debug.Print InputBox("Geben Sie einen Text ein.", "Texteingabe", "[Hier eingeben]")
```

Die **Debug.Print**-Anweisung gibt den zurückgelieferten Wert im Direktfenster des VBA-Editors aus. Wie Sie sehen, verwenden wir hier drei Parameter:

- **Prompt**: Text, der den Benutzer informiert, welche Eingabe von ihm erwartet wird
- **Title**: Text für die Titelleiste des Fensters (optional)
- **Default**: Text, der als Vorgabe im Textfeld angezeigt werden soll (optional)

Die InputBox selbst sieht dann wie in Bild 1 aus. Hier gibt es vier Steuerelemente: das Bezeichnungsfeld für den Text aus dem Parameter **Prompt**, das Textfeld zur Eingabe des gesuchten Wertes durch den Benutzer, das gegebenenfalls den Wert aus **Default** anzeigt, sowie die beiden Schaltflächen **OK** und **Abbrechen**.

Wenn der Benutzer die **OK**-Schaltfläche betätigt, soll die Funktion **InputBox** den aktuell im Textfeld befindlichen Text zurückliefern, bei Betätigung der **Abbrechen**-Schaltfläche soll eine leere Zeichenfolge zurückgegeben werden.

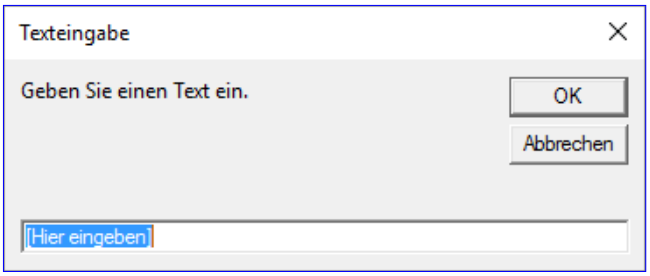


Bild 1: InputBox unter VBA

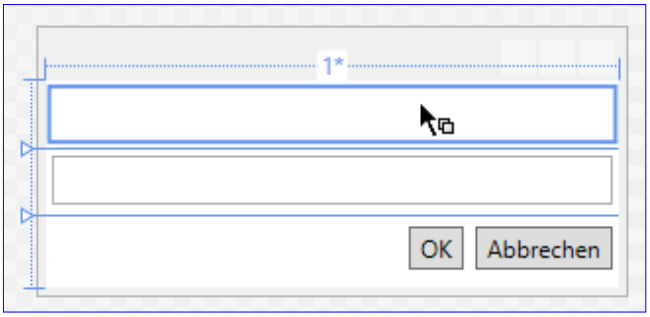


Bild 2: Fenster zur Anzeige einer InputBox im Entwurf

Wenn der Benutzer die Eingabetaste drückt, löst er die Funktion der **OK**-Schaltfläche aus, die **Escape**-Taste bewirkt selbiges für die **Abbrechen**-Schaltfläche.

InputBox als Fenster

Um ein Projekt mit einem Fenster auszustatten, das als InputBox dient, fügen Sie diesem einfach ein normales Objekt des Typs **Fenster (WPF)** hinzu. Nennen Sie das Fenster im Dialog **Neues Element hinzufügen** schlicht **InputBox**. Das Fenster statten Sie mit den Steuerelementen aus Bild 2 aus (das Bezeichnungsfeld haben wir markiert, damit man es überhaupt erkennt).

```
<Window x:Class="InputBoxImEigenbau.InputBox"
        xmlns="http://schemas.microsoft.com/winfx/2006/xaml/presentation"
        xmlns:x="http://schemas.microsoft.com/winfx/2006/xaml"
        xmlns:d="http://schemas.microsoft.com/expression/blend/2008"
        xmlns:mc="http://schemas.openxmlformats.org/markup-compatibility/2006"
        xmlns:local="clr-namespace:InputBoxImEigenbau"
        mc:Ignorable="d"
        Title="{Binding Titel}" Height="136" Width="300" WindowStartupLocation="CenterScreen">
    <Grid>
        <Grid.ColumnDefinitions>
            <ColumnDefinition></ColumnDefinition>
        </Grid.ColumnDefinitions>
        <Grid.RowDefinitions>
            <RowDefinition Height="33"></RowDefinition>
            <RowDefinition Height="33"></RowDefinition>
            <RowDefinition Height="33"></RowDefinition>
        </Grid.RowDefinitions>
        <Label x:Name="lblBezeichnung" Grid.Row="0" Margin="3"></Label>
        <TextBox x:Name="txtEingabe" Grid.Row="1" Margin="3" Height="25"></TextBox>
        <StackPanel Orientation="Horizontal" Grid.Row="2" HorizontalAlignment="Right">
            <Button x:Name="btnOK" Height="22" Padding="5,2,5,2" Margin="3" Content="OK" Click="btnOK_Click"
                IsDefault="True"></Button>
            <Button x:Name="btnCancel" Height="22" Padding="5,2,5,2" Margin="3" Content="Abbrechen"
                Click="btnCancel_Click" IsCancel="True"></Button>
        </StackPanel>
    </Grid>
</Window>
```

Listing 1: Code des WPF-Fensters, das als InputBox dienen soll

Der XAML-Code des Fensters sieht wie in Listing 1 aus.

Hier stecken einige Besonderheiten: Das Fenster soll zentriert erscheinen, also erhält das **Window**-Element den Wert **CenterScreen** für die Eigenschaft **WindowStartupLocation**.

Die Steuerelemente haben wir in einem Grid arrangiert. In der letzten Zeile des Grids haben wir außerdem ein **StackPanel** mit dem Wert **Horizontal** für das Attribut **Orientation** eingebaut, damit wir darin die Schaltflächen nebeneinander anordnen können. Die Schaltfläche **cmdOK** erhält für die Eigenschaft **IsDefault** den Wert **true**, damit sie beim Betätigen der Eingabetaste ausgelöst wird. **cmdAbbrechen** statten wir mit dem Wert **true** für das Attribut **IsCancel** aus, so löst das Betätigen der Taste **Escape** diese Schaltfläche aus.

Code hinter dem InputBox-Fenster

Die Code behind-Datei zu unserer InputBox finden Sie in Listing 2. Die Konstruktor-Methode namens **InputBox** erwartet die Übergabe dreier Parameter namens **Titel**, **Bezeichnung** und **Standardwert**. Sie initialisiert das Fenster und trägt dann den Wert des Parameters **Bezeichnung** für das Attribut **Content** des Elements **lblBezeichnung** ein. Der Wert des Parameters **Titel** wird zunächst überprüft. Wenn der Aufruf diesen Parameter nicht enthält, erhält dieser zunächst als Standardwert eine leere Zeichenfolge. Die Methode prüft dann, ob **Titel** leer ist, und ersetzt diesen Wert dann durch den aktuellen Anwendungstitel, der mit **System.AppDomain.CurrentDomain.FriendlyName** ermittelt wird. Danach landet der Inhalt von **Titel** im Attribut **Title** des **Window**-Elements, das mit **this** referenziert wird. Der Wert des Parameters

```
using System.Windows;

namespace InputBoxImEigenbau {
    public partial class InputBox : Window {
        string eingabe;
        public InputBox(string Bezeichnung, string Titel = "", string Standardwert="") {
            InitializeComponent();
            this.lblBezeichnung.Content = Bezeichnung;
            if (Titel=="") {
                Titel = System.AppDomain.CurrentDomain.FriendlyName;
            }
            this.Title = Titel;
            this.txtEingabe.Text = Standardwert;
            this.txtEingabe.SelectAll();
            this.txtEingabe.Focus();
        }
        public string Eingabe
        {
            set { eingabe = value; }
            get { return eingabe; }
        }
        private void btnCancel_Click(object sender, RoutedEventArgs e) {
            eingabe = "";
            this.Visibility = Visibility.Hidden;
        }

        private void btnOK_Click(object sender, RoutedEventArgs e) {
            eingabe = this.txtEingabe.Text;
            this.Visibility = Visibility.Hidden;
        }
    }
}
```

Listing 2: Code behind-Modul des Fensters **InputBoxImEigenbau**

Standardwert wird schließlich in die Eigenschaft **Text** des Textfeldes **txtEingabe** eingesetzt.

Das Textfeld erfährt noch eine Sonderbehandlung: Die Methode **SelectAll** markiert den kompletten Inhalt und die Methode **Focus** verschiebt den Fokus auf dieses Steuerelement.

Um das Ergebnis zu betrachten, benötigen wir jedoch noch eine Methode, die das **InputBox**-Fenster anzeigt. Diese fügen wir einer neuen Schaltfläche des Fensters **MainWindow** hinzu. Die Methode lautet wie folgt:

```
private void btnInputBox_Click(object sender,
        RoutedEventArgs e) {
    InputBox inputBox = new InputBox("Bezeichnung",
        "Beispieltitel", "Standardwert");
    inputBox.ShowDialog();
    MessageBox.Show(inputBox.Eingabe);
    inputBox.Close();
}
```

Die Methode erstellt ein neues Objekt auf Basis der Klasse **InputBox** und übergibt der Konstruktor-Methode dabei gleich

Ribbons mit WPF

Der geneigte Access-Entwickler weiß mit dem Begriff Ribbon natürlich etwas anzufangen. Aber gibt es so etwas auch für .NET-Anwendungen? Ist das nicht ein reines Office-Feature? Immerhin gibt es ja auch in der Benutzeroberfläche von Visual Studio kein Ribbon, sondern eine Menüleiste und Symbolleisten, wie Sie sie von älteren Office-Versionen bis Office 2003 kennen (man munkelt, einige Entwickler wünschten sich die Menüleisten zurück ...). In diesem Artikel schauen wir uns an, wie weit es mit dem Ribbon unter .NET und speziell unter WPF bestellt ist und welche Unterstützung uns Visual Studio für die Programmierung von Ribbons bietet.

Um herauszufinden, ob es unter WPF Ribbons gibt und wie diese programmiert werden, starten wir mit einem neuen Projekt unter dem Namen **RibbonsInWPF** mit der Vorlage **Visual C#|Windows|WPF-Anwendungen**. Dort beginnen wir mit dem Fenster **MainWindow**, das ja automatisch als Startfenster angelegt wird.

Die schnelle Eingabe des Suchbegriffs Ribbon in das Suchfenster der Toolbox von Visual Studio liefert schon einmal keine Ergebnisse. Also werfen wir einen Blick in den Dialog **Neues Element hinzufügen**, den Sie mit dem gleichnamigen Befehl des Eintrags **Projekt** der Menüleiste öffnen. Auch hier gibt es eine Suchmöglichkeit, die wir allerdings ebenfalls ohne Erfolg nutzen. Gibt es etwa gar keine Ribbons unter WPF?

Vielleicht müssen wir dafür eine spezielle Bibliothek hinzufügen. Also öffnen wir den **Verweis-Manager** (Menüeintrag **Projekt|Verweis hinzufügen...**) und suchen hier nach dem Ausdruck **Ribbon**. Bingo! Hier werden wir endlich fündig und fügen den Eintrag aus Bild 1 per Mausklick zu den eingebundenen Referenzen hinzu. In den Steuerelementen und auch im Dialog **Neues Element hinzufügen** finden sich allerdings weiterhin keine neuen Elemente. Um es kurz zu machen: Auch in Visual Studio müssen Sie manuell XML-Code eingeben, um ein Ribbon zu definieren. Also legen wir los und fügen im **Grid**-Element ein erstes **Ribbon**-Element hinzu:

```
<Window x:Class="RibbonMitWPF.MainWindow">
    <Grid>
        <Ribbon></Ribbon>
    </Grid>
</Window>
```

Das Ergebnis bekommen wir allerdings immerhin direkt in Form eines entsprechenden Elements im Entwurf des WPF-

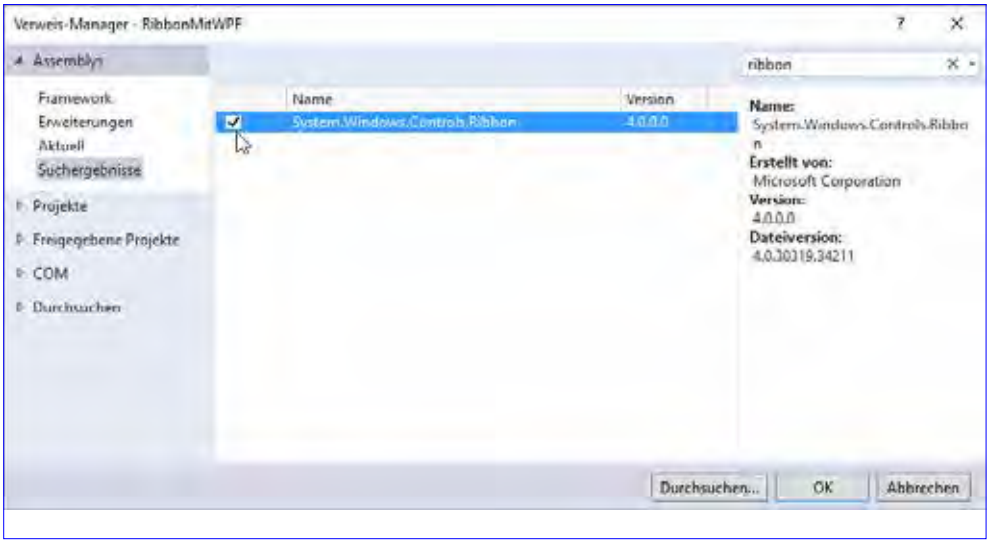


Bild 1: Hinzufügen einer Referenz auf die Bibliothek **System.Windows.Controls.Ribbon**

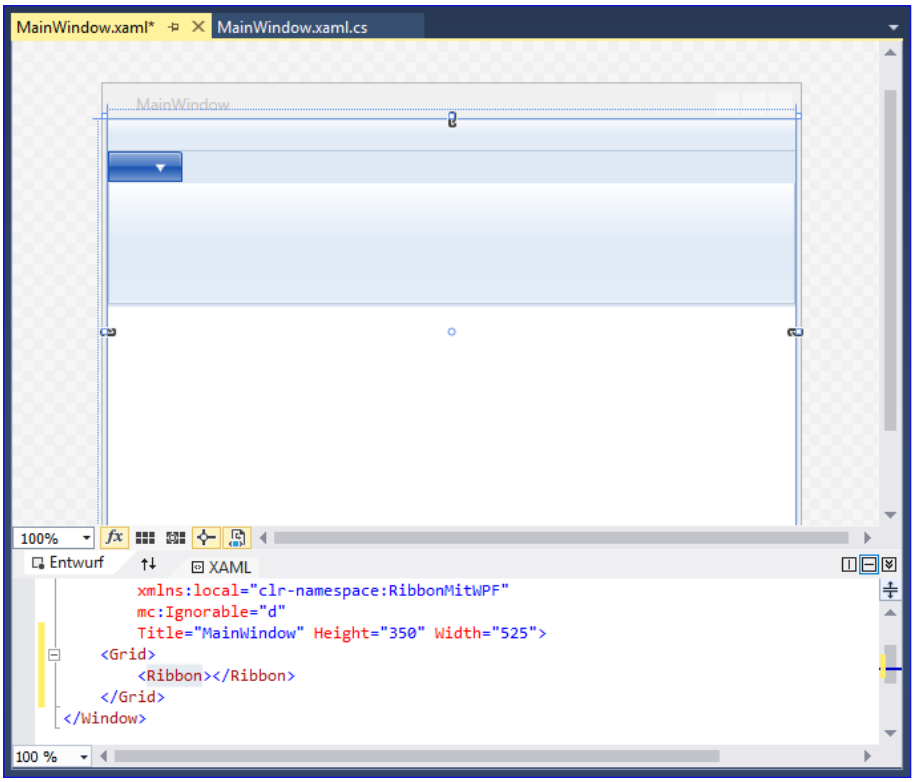


Bild 2: Hinzufügen eines leeren Ribbons

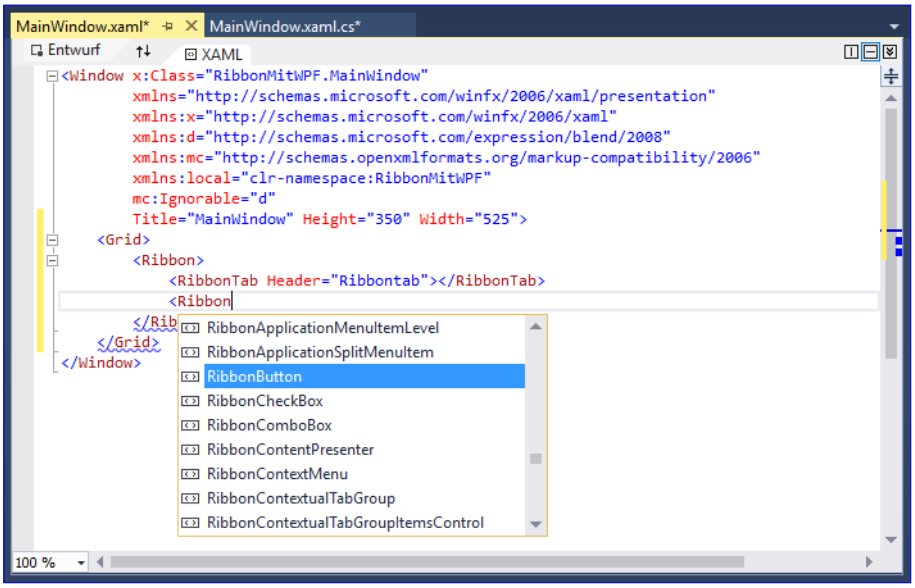


Bild 3: Zugriff auf die Ribbon-Elemente per IntelliSense

Fensters zu sehen (siehe Bild 2). Ein Klick mit der rechten Maustaste auf das Ribbon liefert ebenfalls keine Möglichkeit, auf die Schnelle ein paar Steuerelemente zum Ribbon hinzuzufügen. Also genießen wir zumindest die Bereitstellung von IntelliSense bei der manuellen Eingabe der notwendigen Elemente.

Tab-Element hinzufügen

Ein Ribbon-Tab legen Sie über das **RibbonTab**-Element an, nicht wie unter Office mit dem **tab**-Element:

```
<Ribbon>
    <RibbonTab
Header="Ribbontab"></RibbonTab>
</Ribbon>
```

Grundsätzlich finden Sie die Ribbon-spezifischen Elemente alle schnell, wenn Sie mit IntelliSense-Unterstützung direkt die Zeichenfolge **Ribbon...** eingeben (siehe Bild 3). Hier tauchen dann auch schnell die bekannten Elemente wie **RibbonButton**, **RibbonGroup** und so weiter auf.

Schaltfläche hinzufügen

Fügen wir als Nächstes eine Schaltfläche hinzu. Wenn wir nicht achtgeben und das **RibbonButton**-Element außerhalb des **RibbonTab**-Elements im XAML-Code hinzufügen, geschieht tatsächlich das, was Sie in Bild 4 sehen: Das **RibbonButton**-Element wird oben neben dem Ribbon-Registerreiter angelegt. Ob und wozu man das braucht, erklärt sich nicht – wir wollen uns aber an den gewohnten Strukturen aus dem Office-Bereich orientieren. Also fügen wir das **RibbonButton**-Element auch nicht einfach in das **RibbonTab**-Element ein, sondern fügen zunächst ein **RibbonGroup**-Element hinzu. Grundsätzlich kann man sich als Access-Entwickler, der sich schon einmal mit der

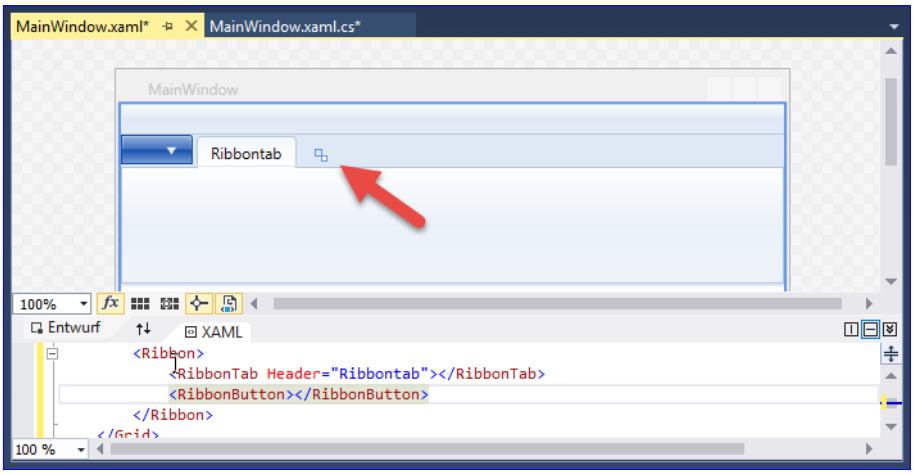


Bild 4: Ein Ribbon-Button außerhalb eines Tab-Elements

```
<Ribbon Title="Beispielribbon">
  <RibbonTab Header="Ribbontab">
    <RibbonGroup Header="Ribbongruppe 1">
      <RibbonButton Label="Beispielbutton 1"></RibbonButton>
      <RibbonButton Label="Beispielbutton 2"></RibbonButton>
      <RibbonButton Label="Beispielbutton 3"></RibbonButton>
    </RibbonGroup>
    <RibbonGroup Header="Ribbongruppe 2">
      <RibbonButton Label="Beispielbutton 4"></RibbonButton>
      <RibbonButton Label="Beispielbutton 5"></RibbonButton>
      <RibbonButton Label="Beispielbutton 6"></RibbonButton>
    </RibbonGroup>
  </RibbonTab>
</Ribbon>
```

Listing 1: Code für ein Ribbon mit zwei Gruppen und einigen Schaltlfächen

Bild 5: Ein Ribbon in einer laufenden Anwendung

Programmierung des Ribbons beschäftigt hat, merken, dass die bekannten Elemente alle mit dem Präfix **Ribbon...** ausgestattet wurden (also **RibbonTab** statt **tab**, **RibbonGroup** statt **group** et cetera).

Ein **RibbonTab**-Element mit zwei **RibbonGroup**- und je drei **RibbonButton**-Elementen definieren wir schließlich wie in Listing 1.

Das Ergebnis nach einem Klick auf die Taste **F5** zum Start des Debuggers zeigt Bild 5. Es gibt ein paar Besonderheiten, die wir an dieser Stelle klären wollen: Zunächst können Sie bereits für das **Ribbon**-Element mit dem Attribut **Title** eine Überschrift festlegen, die dann in der Abbildung beispielsweise **Beispielribbon** lautet. Dies ist unter Office so nicht möglich. Die zweite Besonderheit ist, dass es für die Beschriftungen so viele verschiedene Attribute gibt. Beim **Ribbon**-Element selbst heißt sie **Title**, beim **RibbonTab** und bei der **RibbonGroup** lautet das Attribut **Header** und bei der Schaltfläche (**RibbonButton**) heißt sie schließlich **Label**. Der Hintergrund sind die verschiedenen Basis-Elemente, von denen die Ribbon-Elemente erben. Wenn man das erstmal herausgefunden hat, gewöhnt man sich allerdings recht schnell daran.

Bilder im WPF-Ribbon

Bevor wir uns um die Funktionalität kümmern, also um die Ereignisse der einzelnen Ribbon-Elemente, wollen wir noch für eine etwas schickere Optik sorgen. Dazu gehören im Ribbon natürlich ein paar Icons. Im Gegensatz zu Access (und den übrigen Office-Anwendungen), wo einiges an Code benötigt wird, um Bilder im Ribbon anzuzeigen, gelingt dies unter WPF relativ einfach. Dazu fügen Sie zunächst die gewünschten Icons zum Projekt hinzu, indem Sie diese einfach in das Projekt im Projektmappen-Explorer ziehen. Bevor Sie damit beginnen, sollten Sie sich überlegen, ob Sie für Bild-

dateien nicht einen eigenen Ordner innerhalb des Projekts anlegen sollen. Dies erledigen Sie über den Kontextmenü-Eintrag **Hinzufügen|Neuer Ordner** des Projekt-Elements. Den neuen Ordner können Sie beispielsweise **Images** nennen. Danach ziehen wir einige Bilddateien aus dem Windows Explorer in dieses Verzeichnis. Sammlungen solcher Icons können Sie im Internet kostenlos finden, für hochwertige Kollektionen wie etwa von **iconexperience.com** müssen Sie jedoch ein paar Euro hinlegen.

Der neue Ordner **Images** mit einigen Bildern sieht dann so wie in Bild 6 aus. Wenn Sie die Einträge mit der Maus überfahren, blendet Visual Studio eine Vorschau der jeweiligen Bilder ein. Nun wollen wir die Bilder auch im Ribbon anzeigen, zunächst als Symbol für die Schaltflächen.

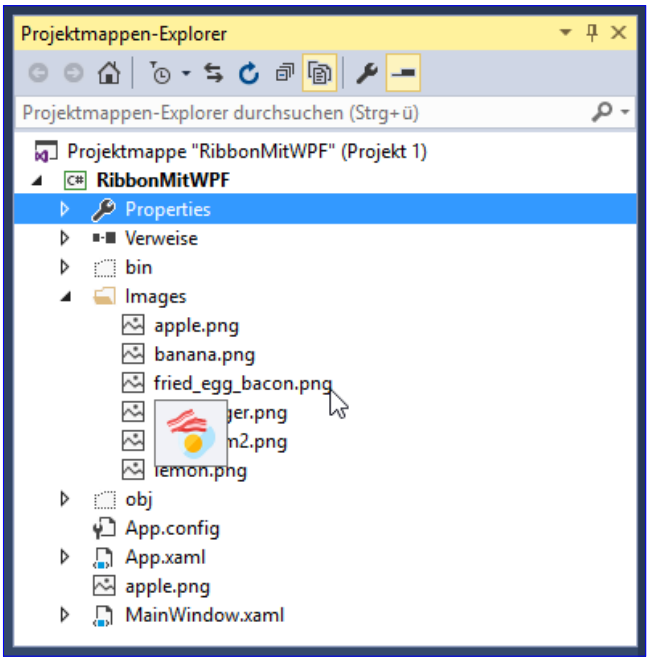


Bild 6: Bilder für die Anzeige im Ribbon im Projekt-Explorer

Dazu fügen wir den Schaltflächen eines der beiden Attribute **SmallImageSource** oder **LargeImageSource** hinzu und legen den Namen der Bilddatei inklusive Verzeichnis als Wert fest. Dies sieht beispielsweise wie folgt aus:

```
<RibbonButton Label="Beispielbutton 1"
    LargeImageSource="Images\apple.png" />
...
<RibbonButton Label="Beispielbutton 4"
   SmallImageSource="Images\ice_cream2.png" />
```

In der Anwendung erscheinen die Bilder in den veschiedenen Größen dann wie in Bild 7. Wir haben hier ausschließlich Bilder der Größe 32x32 verwendet. Üblicherweise verwendet man für das Attribut **LargeImageSource** Bilddateien dieser Größe, für das Attribut **SmallImageSource** nutzt man Bilder der Größe 16x16. Die Bilder werden aber bei Bedarf herunterskaliert. Wenn Sie über eine Icon-Sammlung mit Icons in diesen beiden Größen verfügen, sollten Sie jeweils die korrekte Größe für die entsprechenden Attribute verwenden.

Dazu müssen Sie dann zum Dateinamen der Elemente im Ordner **Images** noch ein Suffix wie etwa **_small** hinzufügen.

Bild 7: Ribbon-Schaltflächen mit einigen Bildern

Ribbon-Titelzeile entfernen

Die Zeile, die durch das Ribbon-Element selbst repräsentiert wird, können Sie mit einem Trick entfernen. Dazu ändern Sie den Typ des Fenster-Objekts von **Window** auf **RibbonWindow**. Dazu sind drei Schritte nötig.

Der erste Schritt ist das Hinzufügen der Klasse mit einer entsprechenden **using**-Anweisung in der Code behind-Klasse des Fensters:

```
using System.Windows.Controls.Ribbon;
```

Danach ändern Sie den Typ der Basisklasse von **MainWindow** von **Window** auf **RibbonWindow**, sodass der Inhalt der Code behind-Klasse nun wie folgt aussieht:

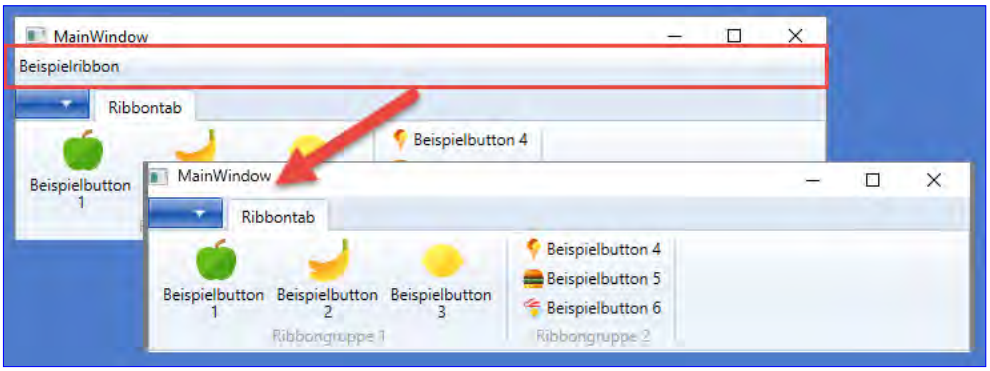


Bild 8: Ribbon mit und ohne Titelleiste

```
namespace RibbonMitWPF {
    public partial class MainWindow : RibbonWindow {
        public MainWindow() {
            InitializeComponent();
        }
    }
}
```

Schließlich ziehen Sie diese Änderung auch im XAML-Code dieser Klasse nach. Dazu ändern Sie das oberste Element ebenfalls von Window in **RibbonWindow**:

```
<RibbonWindow ... Title="MainWindow" Height="350"
Width="625">
    <Grid>
    ...
```

Wenn Sie die vorherige Ansicht und die aktuelle Version vergleichen, stellen Sie fest, dass die Zeile mit dem Text **Beispielribbon** weggefallen ist (siehe Bild 8). Übrigens: Wenn Sie das **Title**-Attribut des **RibbonWindow**-Elements in diesem Fall leer lassen, zeigt der Titel den Inhalt der Eigenschaft **Header** des **Ribbon**-Elements an.

Das Icon der Titelleiste des Fensters können Sie in diesem Fall durch Setzen des Attributs **WindowIconVisibility** auf den Wert **Collapsed** verschwinden lassen:

```
<Ribbon Title="Beispielribbon"
    WindowIconVisibility="Collapsed">
```

Schnellzugriffsleiste nutzen

Warum aber sollte man diesen Bereich überhaupt erhalten? Nun: Wenn Sie Access-Ribbons gewohnt sind, wissen Sie, dass dort oben in der Regel die Schnellzugriffsleiste untergebracht ist, die einige Standardbefehle wie **Wiederholen** oder **Rückgängig** enthält. Sie könnten also durchaus auch selbst einige solcher Befehle dort unterbringen, die immer sichtbar sind – unabhängig davon, welches **RibbonTab**-Element gerade aktiviert ist. Um Elemente zur Schnellstartleiste hinzuzufügen, legen Sie unterhalb des **Ribbon**-Elements ein Element namens **Ribbon.QuickAccessToolbar** an, dem Sie wiederum das Element **RibbonQuickAccessToolbar** unterordnen. Darin finden schließlich die gewünschten Steuerelemente Platz (siehe Bild 9):

```
<Ribbon Title="Beispielribbon">
  <Ribbon.QuickAccessToolBar>
    <RibbonQuickAccessToolBar >
      <RibbonButton SmallImageSource="Images\apple.png">
      </RibbonButton>
      <RibbonButton SmallImageSource="Images\banana.png">
      </RibbonButton>
      <RibbonButton SmallImageSource="Images\lemon.png">
      </RibbonButton>
    </RibbonQuickAccessToolBar>
  </Ribbon.QuickAccessToolBar>
  ...
</Ribbon>
```

Das Anwendungsmenü

Das blaue Element links oben im Ribbon mit dem Pfeil nach unten kennen Sie vermutlich von Office ab Version 2010. Dort können Sie damit den Backstage-Bereich öffnen, wo Sie etwa Zugriff auf die Optionen der jeweiligen Anwendung erhalten. Unter WPF können Sie dieses Element natürlich nach Ihren

Wünschen gestalten. Es heißt **RibbonApplicationMenu** und wird ebenfalls wie das **QuickAccessToolbar**-Element erst mit dem Element **Ribbon.ApplicationMenu** und dann mit **RibbonApplicationMenu** unterhalb des **Ribbon**-Elements integriert.

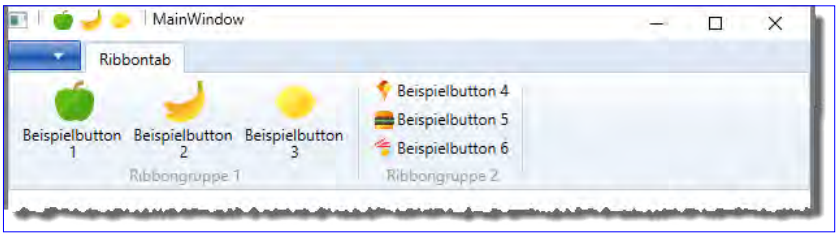


Bild 9: Ribbon mit Elementen in der Schnellzugriffsleiste

Der folgende XAML-Code liefert ein Beispiel für den Einsatz eines minimalen **RibbonApplicationMenu**-Elements. Es enthält lediglich einige Elemente des Typs **RibbonApplicationMenuItem** mit dem Text **Menu Item 1** und so weiter. Außerdem haben wir unsere im Projekt gespeicherten Bilder wieder eingesetzt – diesmal allerdings für das Attribut **ImageSource** (hier gibt es kein **SmallImageSource** oder **LargeImageSource**):

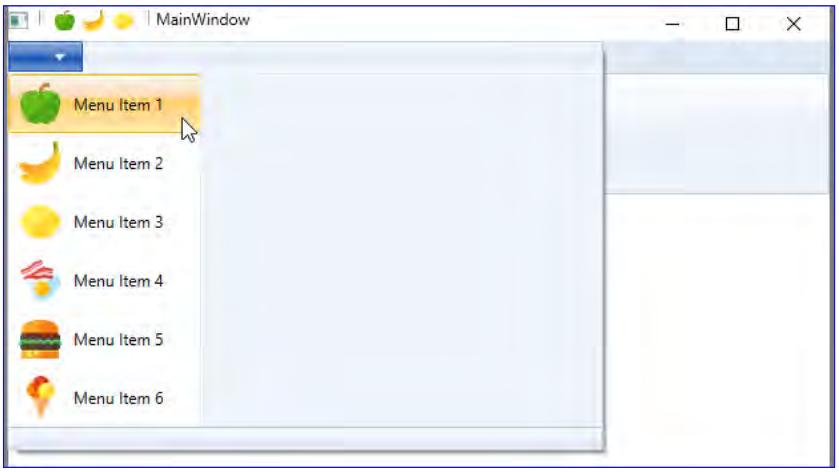


Bild 10: Ribbon mit Elementen im Anwendungsmenü

```
<Ribbon Title="Beispielribbon" >
    <Ribbon.ApplicationMenu>
        <RibbonApplicationMenu>
            <RibbonApplicationMenuItem
                Header="Menu Item 1"
                ImageSource="Images\apple.png" />
            <RibbonApplicationMenuItem
                Header="Menu Item 2"
                ImageSource="Images\banana.png" />
            <RibbonApplicationMenuItem
                Header="Menu Item 3"
                ImageSource="Images\lemon.png" />
            <RibbonApplicationMenuItem
                Header="Menu Item 4"
                ImageSource="Images\fried_egg_bacon.png" />
            <RibbonApplicationMenuItem
                Header="Menu Item 5"
                ImageSource="Images\hamburger.png" />
            <RibbonApplicationMenuItem
                Header="Menu Item 6"
                ImageSource="Images\ice_cream2.png" />
        </RibbonApplicationMenu>
    </Ribbon.ApplicationMenu>
    ...
</Ribbon>
```

Das Ergebnis sieht wie in Bild 10 aus. Wir können den einzelnen Elementen noch Untereinträge hinzufügen:

```
<RibbonApplicationMenuItem Header="Menu Item 1"
        ImageSource="Images\apple.png">
    <RibbonApplicationMenuItem Header="Menu Item 1-1"
        ImageSource="Images\banana.png"/>
    <RibbonApplicationMenuItem Header="Menu Item 1-2"
        ImageSource="Images\lemon.png" />
</RibbonApplicationMenuItem>
```

Gegebenenfalls taucht hier ein Fehler auf (**Der Index "0" befindet sich außerhalb des gültigen Bereichs der PathParameters-Liste mit der Länge "0".**). Diesen können Sie ignorieren. Der Code liefert nun das Ergebnis aus Bild 11.

EDM: Einfaches Detailfenster

Wie Sie ein DataGrid-Steuerelement mit den Daten aus einem Entity Data Model füttern, haben Sie bereits im Artikel »Einführung in das Entity Framework« erfahren. Allerdings möchten Sie gelegentlich auch in den Datensätzen einer Tabelle blättern, die enthaltenen Daten ändern oder neue Datensätze hinzufügen – ganz, wie es unter Access in einem einfachen Detailformular möglich war. Dieser Artikel zeigt, wie Sie dies mit den eingebauten Funktionen von Visual Studio einfach bewerkstelligen können.

Das Ziel dieses Artikels ist es also, ein neues Fenster zu erstellen, das über das Entity Data Model an eine Tabelle gebunden ist – in diesem Fall an die Tabelle **Kunden** unserer Beispieldatenbank namens **Bestellverwaltung** – und das die Daten jeweils eines Datensatzes in entsprechenden Steuerelementen anzeigt. Außerdem wollen wir dem Fenster einige Schaltflächen hinzufügen, mit denen der Benutzer zum ersten, vorherigen, nächsten und letzten und zu einem neuen Datensatz springen kann. Schließlich soll der Benutzer auch Änderungen durchführen und diese speichern können.

Wir verwenden dabei ein neues Projekt des Typs **Visual C#\WPF-Anwendung**. Anschließend erstellen Sie nach bewährter Art ein Entity Data Model mit den Daten der zu verwendenden Datenbank.

Fügen Sie also ein neues Element des Typs **ADO.NET Entity Data Model** zum Projekt hinzu und nennen Sie es **BestellverwaltungModel**. Im folgenden Dialog behalten Sie den Eintrag **EF Designer aus Datenbank** bei. Im nächsten Dialog klicken Sie auf **Neue Verbindung** und wählen dort **Microsoft SQL Server** aus.

Im Dialog Verbindungseigenschaften geben Sie **(localdb)\MSSQLLocalDB** als Servername und **Bestellverwaltung** unter **Mit Datenbank verbinden** ein (siehe Bild 1). Wir gehen an dieser Stelle davon aus, dass Sie die Bestellverwaltung-Datenbank bereits an den SQL Server oder LocalDB angehängt haben (dies erledigen Sie am einfachsten über den SQL Server Management Studio).

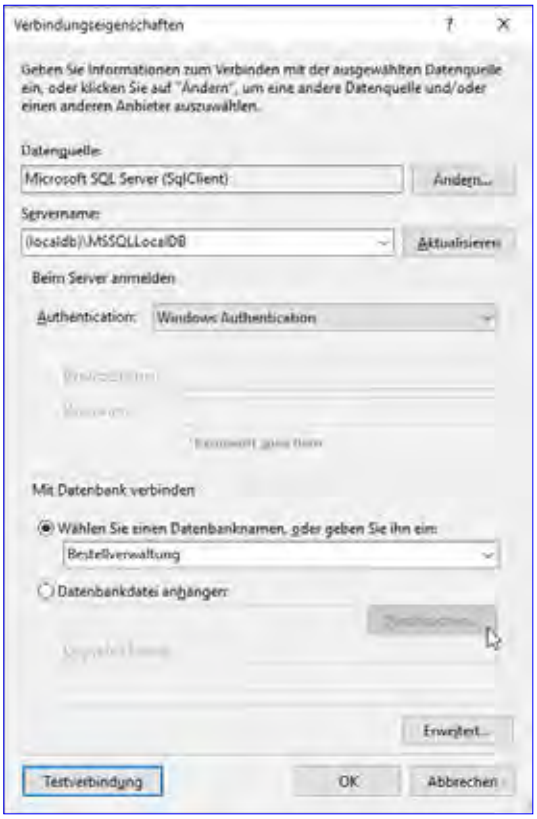


Bild 1: Hinzufügen der Verbindung

Kehren Sie zum vorherigen Dialog zurück und klicken Sie auf **Weiter**. Die Verbindungseinstellungen wollen wir unter dem Namen **BestellverwaltungEntities** speichern. Behalten Sie im folgenden Schritt den Wert **Entity Framework 6.x** bei.

Unter **Wählen Sie Ihre Datenbankobjekte und Einstellungen** klicken Sie einfach den Eintrag **Tabellen** an. Damit fügen Sie alle Tabellen zum Entity Data Model hinzu. Unter **Modelnamespace:** können Sie den Wert **BestellverwaltungModel** beibehalten.

Danach erstellt Visual Studio das Entity Data Model und zeigt dieses in einem Diagramm an. Ändern Sie hier noch die Namen der Klassen vom Plural in den Singular, also etwa von **Kategorien** in **Kategorie**.

Nun erstellen Sie das Projekt erstmalig mit dem Menüeintrag **Erstellen\|<Projektname> erstellen**.

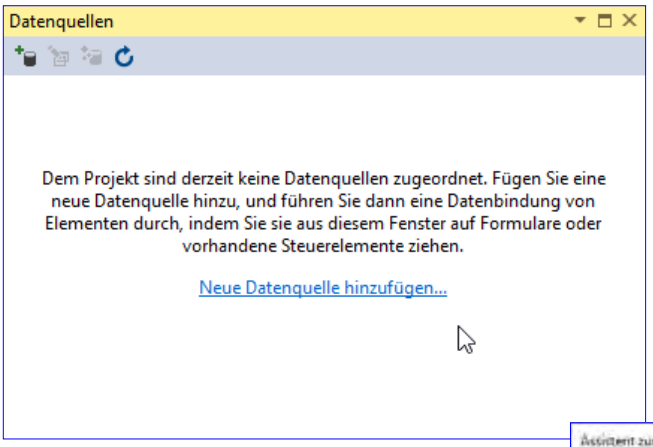


Bild 2: Noch keine Datenquellen vorhanden?

Felder zum Fenster hinzufügen

Nun schauen wir uns den einfachsten Weg an, um die Felder einer Klasse des Entity Data Models zu einem Fenster hinzuzufügen – und zwar zum Fenster **MainWindow.xaml**. Dies wollen wir dazu einfach aus einer entsprechenden Liste in das Fenster ziehen – ähnlich, wie es bei der Feldliste beim Formularentwurf unter Access geschieht.

Um dies zu erledigen, müssen wir zunächst auf Basis des Entity Data Models eine Datenquelle erstellen. Dazu aktivieren Sie, sofern noch nicht geschehen, den Bereich **Datenquellen**, und zwar über den Menübefehl **Ansicht|Weitere Fenster|Datenquellen**. Der Dialog erwartet Sie dann wie in Bild 2.

Hier benötigen wir nun eine Datenquelle. Dazu klicken Sie auf den Link **Neue Datenquelle hinzufügen...** und öffnen so den Assistenten zum Konfigurieren von Datenquellen. Dieser bietet vier mögliche Quellen für die Daten an, von denen wir den Eintrag **Objekt** auswählen (siehe Bild 3).

Im folgenden Schritt werden dann die Objekte des Projekts angezeigt. Unter dem Projektnamen finden Sie einen weiteren Eintrag gleichen Namens, der dann schließlich die Klassen des Projekts anbietet, unter anderem eben auch die Klassen des Entity Data Models, also **Kunde**, **Artikel**, **Bestellung** et cetera. Hier wählen Sie nun wie in Bild 4 den Eintrag **Kunde** aus.

Nach einem Klick auf die Schaltfläche **Fertigstellen** erscheinen die Felder der Klasse **Kunde** im **Datenquellen**-Dialog (siehe Bild 5).

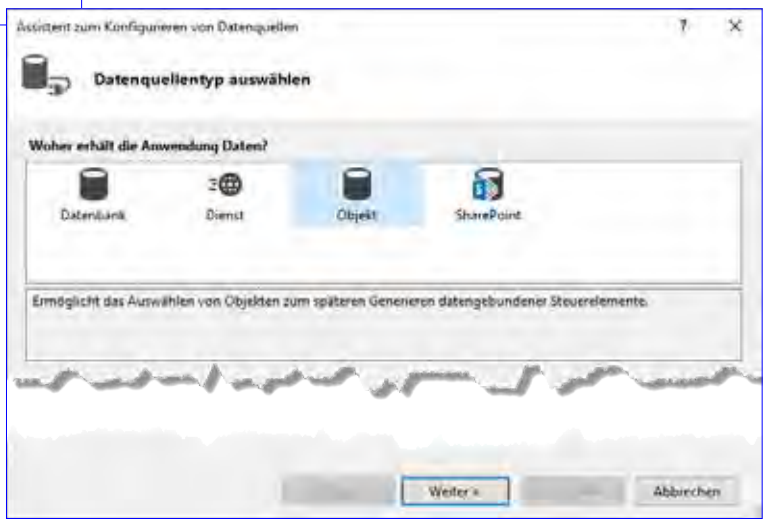


Bild 3: Die Daten sollen aus einem Objekt bezogen werden.

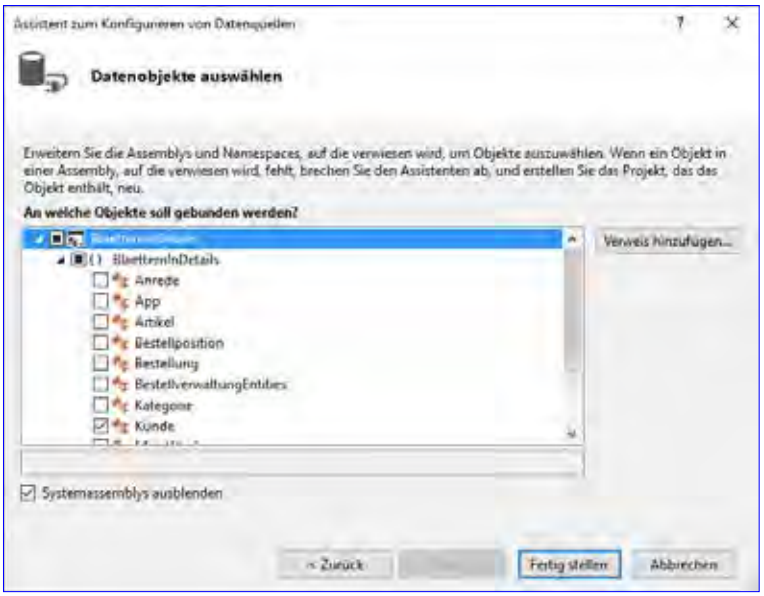


Bild 4: Auswählen des Quellobjekts, hier namens **Kunde**

Damit haben wir uns außerdem einen neuen Eintrag im neuen Ordner **DataSources** im Bereich **Properties** des Projektmappen-Explorers geholt – und zwar namens **Kunde.datasource** (siehe Bild 6).

Felder per Drag and Drop

Wenn Sie nun die Entwurfsansicht des Fensters **MainWindow.xaml** aktivieren, ändert sich die Ansicht im Bereich **Datenquellen** ein wenig (siehe Bild 7). Sie können nun per Mausklick auf die einzelnen Einträge jeweils ein Kombinationsfeld aktivieren, über das sich Feineinstellungen für das Hinzufügen der einzelnen Felder zum Fenster vornehmen lassen.

Die für unseren Fall wichtigste Feineinstellung, wenn wir in einem Rutsch alle Felder in das Fenster **MainWindows.xaml** ziehen wollen, ist die Auswahl der zu erstellenden Steuerelemente (siehe Bild 8). Mit **DataGrid** erstellen Sie ein DataGrid mit allen gewünschten Feldern. **ComboBox** erstellt ein Kombinationsfeld, was in diesem Szenario jedoch Nacharbeiten erfordert. **Liste** erstellt ein Listenfeld mit den Daten der Datenquelle. Und **Detail** erstellt schließlich die gewünschte Ansicht – daher wählen wir diesen Eintrag aus, was dazu führt, dass links neben dem Eintrag **Kunde** ein entsprechendes Icon erscheint. Nun ziehen Sie einfach den kompletten **Kunde**-Knoten in den Entwurf des Fensters **MainWindow.xaml**. Das Ergebnis sieht wie in Bild 9 aus.

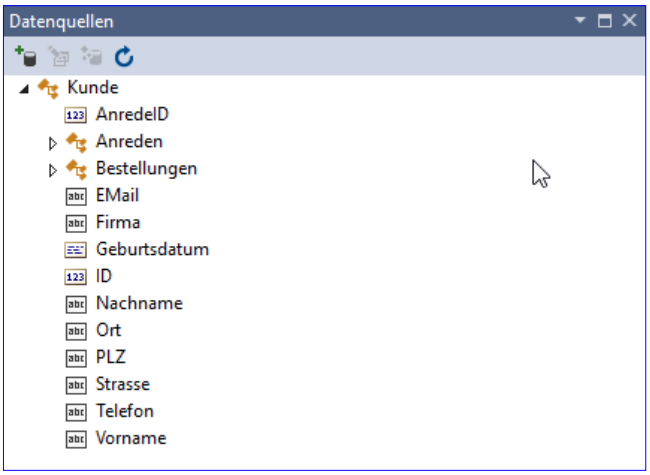


Bild 5: Nun steht eine Datenquelle zur Verfügung.

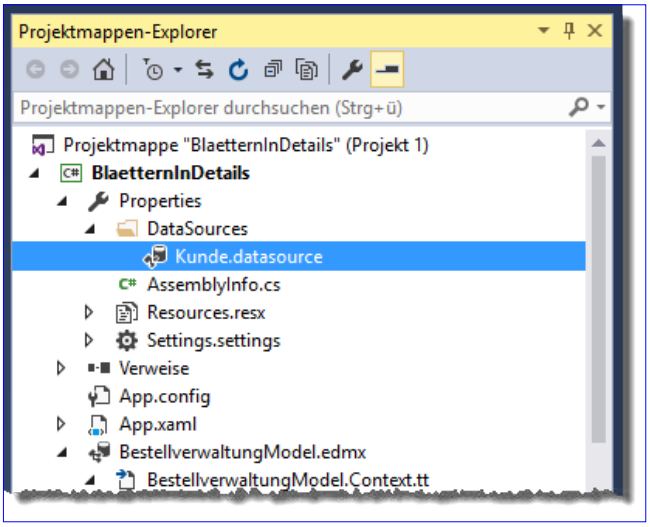


Bild 6: Die Datenquelle im Projektmappen-Explorer

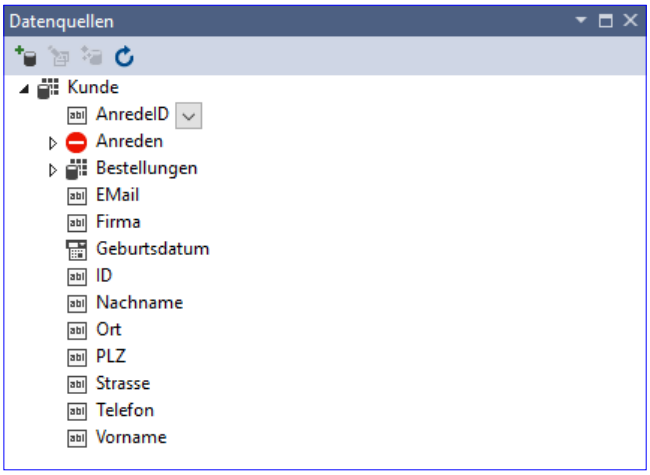


Bild 7: Geänderte Ansicht bei gleichzeitiger Anzeige einer **.xaml**-Datei

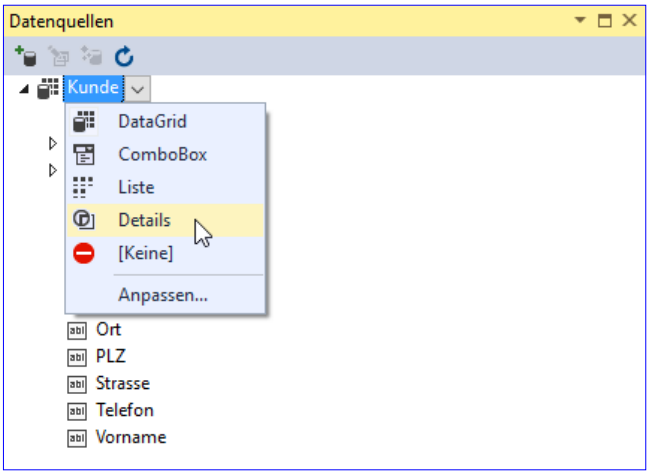


Bild 8: Festlegen der Ansicht

Wenn wir nun das Projekt starten, erhalten wir zwar das Fenster mit den Feldern in der noch unsortieren Anordnung und ohne Kombinationsfelder etwa für das Feld **AnredeID**, aber es zeigt keine Daten an (siehe Bild 10). Was nun?

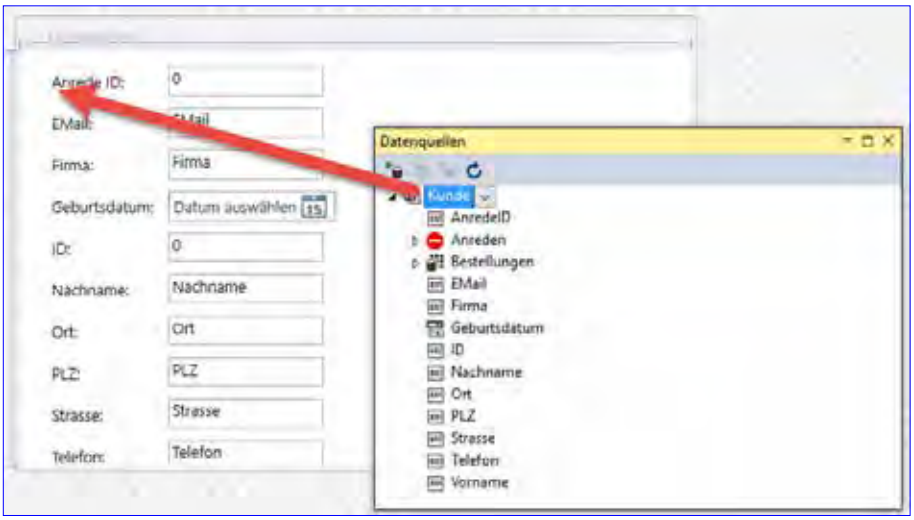


Bild 9: Detailfenster per Drag and Drop

Schauen wir uns zunächst an, was wir durch das Hinzufügen der Steuerelemente per Drag and Drop überhaupt für Änderungen durchgeführt haben – beziehungsweise welche Änderungen automatisch erledigt wurden. Dass für jedes Feld der Datenquelle jeweils ein Steuerelement zum Fenster hinzugefügt wurde, haben Sie ja bereits gesehen. Dementsprechend hat auch eine Menge Code Einzug in das **.xaml**-Dokument **MainWindow.xaml** genommen.

Im oberen Teil wurde ein Eigenschaftselement namens **Windows.Resources** mit folgendem Inhalt eingefügt:

```
<Window.Resources>
    <CollectionViewSource x:Key="kundeViewSource" d:D
esignSource="{d:DesignInstance {x:Type local:Kunde},
CreateList=True}"/>
</Window.Resources>
```

Das **CollectionViewSource**-Element ist eine Verbindung zwischen der Benutzeroberfläche und den Daten. Es erhält automatisch über das Attribut **key** einen Schlüssel namens **kundeViewSource**, damit man es später in den Elementen der Benutzeroberfläche referenzieren kann. Der hintere Teil enthält Informationen über das Aussehen der gebundenen Steuerelemente zur Entwurfszeit.

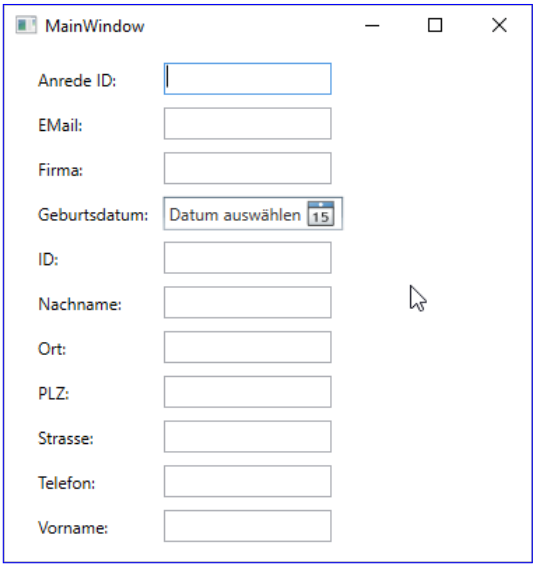


Bild 10: Detailfenster noch ohne Daten

Wenn Sie den Teil **d:DesignSource="{d:DesignInstance {x:Type local:Kunde}, CreateList=True}"** entfernen, werden Sie den Unterschied erkennen.

Schauen wir weiter nach unten, sehen wir ein **Grid**-Element, dessen wichtigste Eigenschaft in diesem Zusammenhang das Attribut **DataContext** ist.

Dieses enthält den Wert **{StaticResource kundeViewSource}** und legt somit fest, dass es die Daten aus der weiter oben festgelegten statischen Ressource beziehen soll (siehe Listing 1). Dies ist die Voraussetzung dafür, dass wir für die im **Grid**-Element enthaltenen Steuerelemente Bindungen festlegen können, über die sie dann auf die Daten der **CollectionViewSource** zugreifen können. Dies gilt für alle Elemente, die dem **Grid**-Element untergeordnet sind.

Im Grid folgen nun einige Zeilen Code, die das Aussehen des Grids festlegen, also wie viele Zeilen und Spalten es enthält und welche Breite beziehungsweise Höhe diese haben (hier in der Regel auf den Wert **Auto** eingestellt).

```
<Grid x:Name="grid1" DataContext="{StaticResource kundeViewSource}" HorizontalAlignment="Left" Margin="204,55,0,0"
        VerticalAlignment="Top">
    <Grid.ColumnDefinitions>
        <ColumnDefinition Width="Auto"/>
        <ColumnDefinition Width="Auto"/>
    </Grid.ColumnDefinitions>
    <Grid.RowDefinitions>
        <RowDefinition Height="Auto"/>
        ...
    </Grid.RowDefinitions>
    <Label Content="Anrede ID:" Grid.Column="0" HorizontalAlignment="Left" Margin="3" Grid.Row="0"
        VerticalAlignment="Center"/>
    <TextBox x:Name="anredeIDTextBox" Grid.Column="1" HorizontalAlignment="Left" Height="23" Margin="3" Grid.Row="0"
        Text="{Binding AnredeID, Mode=TwoWay, NotifyOnValidationError=true, ValidatesOnExceptions=true}"
        VerticalAlignment="Center" Width="120"/>
    <Label Content="EMail:" ... />
        <TextBox x:Name="eMailTextBox" ...
            Text="{Binding EMail, Mode=TwoWay, NotifyOnValidationError=true, ValidatesOnExceptions=true}" ... />
    <Label Content="Firma:" ... />
    <TextBox x:Name="firmaTextBox"
        Text="{Binding Firma, Mode=TwoWay, NotifyOnValidationError=true, ValidatesOnExceptions=true}" ... />
    <Label Content="Geburtsdatum:" ... />
    <DatePicker x:Name="geburtsdatumDatePicker" ...
        SelectedDate="{Binding Geburtsdatum, Mode=TwoWay, NotifyOnValidationError=true, ValidatesOnExceptions=true}" ... />
    <Label Content="ID:" Grid.Column="0" HorizontalAlignment="Left" Margin="3" Grid.Row="4" VerticalAlignment="Center"/>
    <TextBox x:Name="iDTextBox" ...
        Text="{Binding ID, Mode=TwoWay, NotifyOnValidationError=true, ValidatesOnExceptions=true}" ... />
    <Label Content="Nachname:" ... />
    <TextBox x:Name="nachnameTextBox" ...
        Text="{Binding Nachname, Mode=TwoWay, NotifyOnValidationError=true, ValidatesOnExceptions=true}" ... />
    <Label Content="Ort:" ... />
    <TextBox x:Name="ortTextBox" ...
    Text="{Binding Ort, Mode=TwoWay, NotifyOnValidationError=true, ValidatesOnExceptions=true}" ... />
    <Label Content="PLZ:" ... />
    <TextBox x:Name="pLZTextBox" ...
        Text="{Binding PLZ, Mode=TwoWay, NotifyOnValidationError=true, ValidatesOnExceptions=true}" ... />
    <Label Content="Strasse:" ... />
    <TextBox x:Name="strasseTextBox" ...
        Text="{Binding Strasse, Mode=TwoWay, NotifyOnValidationError=true, ValidatesOnExceptions=true}" ... />
    <Label Content="Telefon:" ... />
    <TextBox x:Name="telefonTextBox" ...
        Grid.Row="9" Text="{Binding Telefon, Mode=TwoWay, NotifyOnValidationError=true, ValidatesOnExceptions=true}" ... />
    <Label Content="Vorname:" ... />
    <TextBox x:Name="vornameTextBox" ...
    Text="{Binding Vorname, Mode=TwoWay, NotifyOnValidationError=true, ValidatesOnExceptions=true}" />
</Grid>
```

Listing 1: Die per Drag and Drop erstellten Steuerelemente samt Datenbindung

Dies erfolgt in den Eigenschaftselementen **Grid.ColumnDefinitions** und **Grid.RowDefinitions**, die wiederum entsprechende **ColumnDefinition**- und **RowDefinition**-Elemente enthalten.

Für jedes Feld der Datenquelle hat Visual Studio jeweils ein **Label**-Element angelegt sowie ein entsprechendes Steuerelement. In diesem Fall handelt es sich nur um **TextBox**- und **DatePicker**-Steuerelemente.

Die **TextBox**-Steuerelemente und auch das **DatePicker**-Steuerelement wurden mit jeweils einem Attribut ausgestattet, das sich auf die Datenbindung bezieht. Für das Feld **EMail** lautet der Code des Attributs **Text** beispielsweise so:

```
Text="{Binding EMail, Mode=TwoWay,
NotifyOnValidationError=true, ValidatesOnExceptions=true}"
```

Die Angabe von **EMail** direkt hinter dem **Binding**-Schlüsselwort heißt, dass das Steuerelement als **Text** den Inhalt des Feldes **EMail** der Datenquelle anzeigen soll. **Mode=TwoWay**: legt fest, dass Änderungen an den Daten an das zugrunde liegende Objekt zurückgegeben werden. Würde hier der Wert **OneWay** stehen, würden sich Änderungen nicht auf die Datenquelle auswirken.

Das hinter **Binding** angegebene Feld bezieht sich auf den **DataContext** im übergeordneten Objekt, der wiederum die statische Ressource **kundeViewSource** referenziert. Diese ist wiederum im Element **Window.Resources** als **CollectionViewSource**-Element definiert, wie wir ja weiter oben bereits gesehen haben.

Daten bereitstellen

Nun wissen wir aber noch nicht, woher diese **CollectionViewSource**, deren **Key**-Attribut den Wert **kundeViewSource** hat, die Daten bezieht. Und genau das erledigen wir in der Code behind-Klasse. Beziehungsweise sollte das beim Drag and Drop der Entität in das Fenster erfolgt sein – was aber nicht wie gewünscht geschehen ist, denn sonst hätten wir ja weiter oben beim ersten Start der Anwendung keine leeren Steuerelemente vorgefunden.

Schauen wir uns also an, was uns der Assistent beschert hat. Das ist gar nicht so viel, wie Listing 2 zeigt. Zunächst einmal: Damit die hier definierte Ereignismethode **Window_Loaded** ausgelöst wird, hat Visual Studio das Attribut **Loaded** mit dem Namen der Methode für das **Window**-Element hinterlegt:

```
<Window x:Class="BlaetternInDetails.MainWindow"
... Title="MainWindow" Height="350" Width="632"
Loaded="Window_Loaded">
```

Die Ereignismethode selbst instanziert dann lediglich ein Objekt namens **kundeViewSource** mit dem Typ **CollectionViewSource** und verbindet es mit der für die Benutzeroberfläche festgelegten Ressource, indem sie diese mit **FindResource** ermittelt und nach einer Konvertierung in den richtigen Datentyp zuweist.

Darunter finden wir dann zwei auskommentierte Zeilen, von denen die zweite der Ansatzpunkt zum Füllen der Benutzeroberfläche mit den gewünschten Daten ist. Wir müssen nur noch **[generische Datenquelle]** durch den richtigen Ausdruck ersetzen.

```
private void Window_Loaded(object sender, RoutedEventArgs e) {
    System.Windows.Data.CollectionViewSource kundeViewSource = ((System.Windows.Data.CollectionViewSource)
                                                                (this.FindResource("kundeViewSource")));
    // Laden Sie Daten durch Festlegen der CollectionViewSource.Source-Eigenschaft:
    // kundeViewSource.Source = [generische Datenquelle]
}
```

Listing 2: Automatisch generierter Code

EDM: Lookup-Kombinationsfelder

Im Artikel »Einfaches Detailfenster« sehen wir uns an, wie Sie per Drag and Drop die Daten einer Kundentabelle über ein Entity Data Model zu einem Fenster hinzufügen und dieses mit Navigationsschaltflächen ausstatten. Eine Kleinigkeit fehlt dort noch: Die Umwandlung des Textfeldes mit den Anreden in ein entsprechendes Kombinationsfeld, mit dem sich die Anreden auswählen lassen. Dies reichen wir im vorliegenden Artikel nach.

In dem genannten Artikel haben wir ein Entity Data Model auf Basis unserer Beispieldatenbank **Bestellverwaltung** erstellt, das unter anderem die Entitäten **Kunde** und **Anrede** enthält. Per Drag and Drop haben wir aus dem Datenquellen-Fenster alle Felder der Entität **Kunde** in Form einer Detailansicht in das Fenster gezogen. Das Fenster haben wir außerdem mit Navigationsschaltflächen und mit Schaltflächen zum Erstellen neuer Datensätze und zum Speichern geänderter und neuer Datensätze ergänzt. Nun wollen wir noch aus dem Textfeld, das bisher statt einer Anrede nur den in der Entität **Kunde** gespeicherten Zahlenwert des Fremdschlüsselfeldes der entsprechenden Tabelle namens **Kunden** anzeigt, ein Kombinationsfeld machen. Dieses soll zwar den Wert des Fremdschlüsselfeldes als gebundenen Wert enthalten, aber den Wert des entsprechenden Feldes der Lookup-Tabelle anzeigen und die übrigen Datensätze zur Auswahl anbieten.

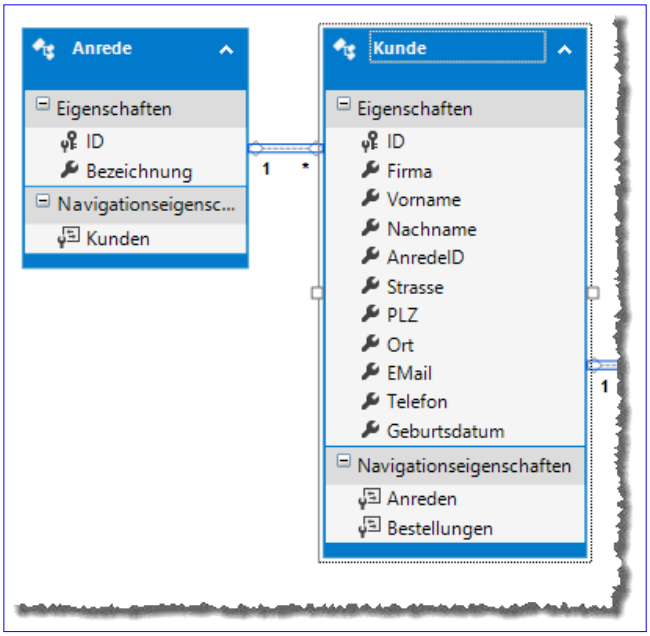


Bild 1: Betroffene Tabellen des Entity Data Models

Die beiden relevanten Tabellen des Entity Data Models sehen Sie in Bild 1. Das Fenster selbst enthält die Felder der Entität **Kunde**. Wir wollen das Steuerelement, das beim Einfügen der Felder per Drag and Drop aus dem Datenquellen-Bereich für das Feld **AnredeID** eingefügt wurde, entfernen und dafür ein Kombinationsfeld hinterlegen.

Wenn wir die Felder wie im Artikel **EDM: Einfaches Detailfenster** einfügen, können wir zumindest

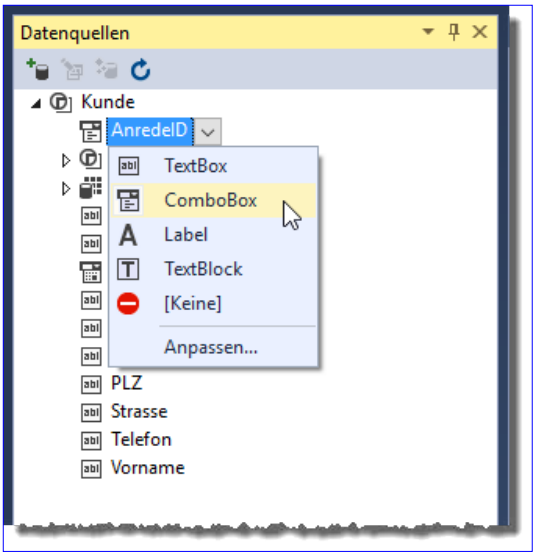


Bild 2: Vorkonfigurieren des Feldes **AnredeID** als Kombinationsfeld

schon einmal den Steuerelementtyp über das entsprechende Kombinationsfeld im Bereich **Datenquellen** einstellen. Dazu wählen Sie dort den Eintrag **ComboBox** aus (siehe Bild 2). Nach dem Sortieren der Steuerelemente wie im oben genannten Artikel beschrieben, fügen Sie noch den Bereich mit den Navigationsschaltflächen hinzu – ebenfalls nach der Vorgehensweise aus dem oben genannten Artikel. Natürlich können Sie auch die

dortigen Beispiele weiterverwenden (die Beispielfenster finden Sie ohnehin in einem einzigen Beispielprojekt).

Erster Versuch

Nachdem wir das Fenster soweit eingerichtet haben, dass es die Daten der zugrunde liegenden Tabelle anzeigt, starten wir das Projekt und schauen uns an, was das Kombinationsfeld uns liefert. Wie Bild 3 zeigt, enthält es wohl die gleiche Datenquelle wie das Fenster selbst und zeigt die Werte des Feldes **AnredeID** für alle Datensätze der Tabelle **Kunden** an.

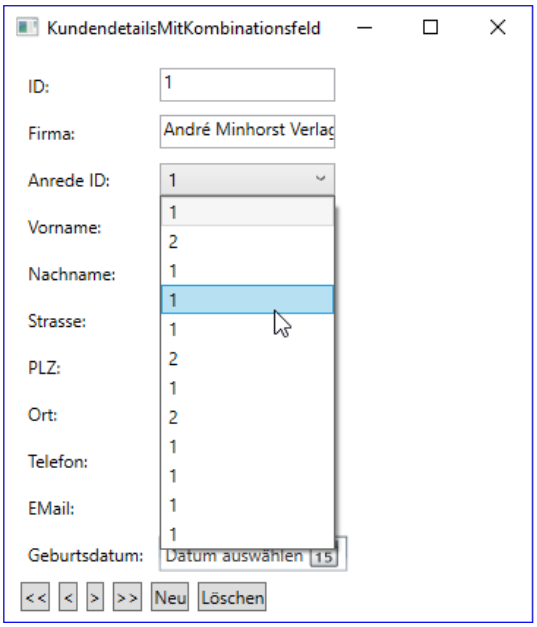


Bild 3: Das Kombinationfeld zeigt lediglich alle Werte, die für die **Kunde**-Entitäten vergeben wurden.

Eigene Datenquelle für das Kombinationsfeld

Der nächste Schritt ist also zunächst, dem Fenster eine eigene Datenquelle für die Einträge der Tabelle **Anrede**, welche das Kombinationsfeld anzeigen soll, hinzuzufügen. Analog zur Variablen **kundeViewSource** des Typs **CollectionViewSource** legen wir eine weitere **CollectionViewSource** an, diesmal namens **anredeViewSource** – dies alles im Kopf der partiellen Klasse im Code behind-Modul des Fensters:

```
CollectionViewSource anredenViewSource;
```

Einige weitere notwendige Schritte nehmen wir in der Ereignismethode **Window_Loaded** vor, die auch schon das **CollectionViewSource**-Objekt **kundeViewSource** gefüllt hat (siehe Listing 1). Hier kommen nun drei Anweisungen hinzu: Die erste weist der Variablen **anredenViewSource** die gleichnamige Ressource hinzu, die wir gleich noch im XAML-Code anlegen müssen. Die zweite lädt mit der **Load**-Methode des **DBSet**-Objekts **Anreden** die Daten der Tabelle **Anreden** aus der Datenbank in das Entity Data Model. Die dritte weist der **Source**-Eigenschaft von **anredenViewSource** die aktuell im Speicher befindlichen Daten aus **DBContext.Anreden** über die **Local**-Eigenschaft zu.

XAML-Code anpassen

Nun fehlen noch die Anpassungen im XAML-Code des Fensters. Als Erstes kümmern wir uns um die Ressource.

Wir haben zwar im Code behind-Modul eingestellt, dass eine CollectionViewSource-Klasse erzeugt und über das Entity Data Model mit den Daten der Tabelle **Anreden** gefüllt wird und angegeben, dass diese mit einer Ressource verknüpft wird, aber diese Ressource ist noch nicht vorhanden.

Diese fügen wir wie in Listing 2 zum Eigenschaftselement **Window.Resources** hinzu, und zwar als **CollectionViewSource**-Element wie das bereits vorhandene Element mit dem Key **kundeViewSource**. Hier verwenden wir für das Attribut **local** jedoch die Entität **Anrede**.

```
private void Window_Loaded(object sender, RoutedEventArgs e) {
    anredenViewSource = ((CollectionViewSource)(this.FindResource("anredenViewSource")));
    kundeViewSource = ((CollectionViewSource)(this.FindResource("kundeViewSource")));
    DBContext.Kunden.Load();
    kundeViewSource.Source = DBContext.Kunden.Local;
    DBContext.Anreden.Load();
    anredenViewSource.Source = DBContext.Anreden.Local;
}
```

Listing 1: Verbinden und füllen des zweiten **CollectionViewSource**-Objekts **anredeViewSource**

EDM: Kombinationsfelder erweitern

Im Artikel »Lookup-Kombinationsfelder« zeigen wir, wie Sie eine per Drag and Drop aus dem Datenquellen-Bereich hinzugefügte Entität mit einem Kombinationsfeld versehen, mit dem Sie die Daten einer Lookup-Tabelle auswählen können. Dies wollen wir nun noch erweitern, und zwar um die Möglichkeit der Eingabe neuer Werte für die Lookup-Tabelle direkt über das Kombinationsfeld. Dazu nutzen wir ein neues Fenster, das die Daten der Tabelle Artikel anzeigt – und deren Kategorie per Kombinationsfeld auswählt.

Neuen Eintrag hinzufügen

Unter Access war es relativ einfach, ein Kombinationsfeld mit einer Funktion auszustatten, die dafür sorgte, dass neu eingegebene Einträge, die noch nicht in der Datensatzherkunft vorhanden waren, zur zugrunde liegenden Tabelle hinzugefügt werden und dann direkt im Kombinationsfeld angezeigt werden konnten. Wir wollen uns anschauen, wie dies unter WPF unter Verwendung unseres Entity Data Models auf Basis der Tabellen der Beispieldatenbank **Bestellverwaltung** funktioniert. Ein passendes Beispiel sind die Tabellen **Artikel** und **Kategorien**. Die Tabelle **Artikel** enthält ein Fremdschlüsselfeld, und zwar zur Auswahl der Kategorie des Artikels. Die Kategorien stammen wiederum aus der Tabelle **Kategorien**.

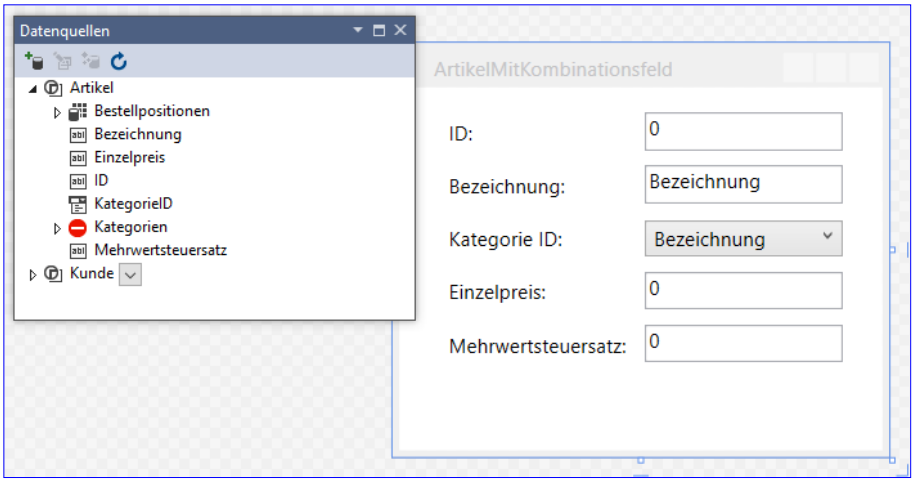


Bild 1: Erstellen des Beispielfensters

Für das neue Beispiel fügen wir der Datenquelle des Projekts die Entität **Artikel** hinzu. Ein neues Fenster soll die Daten der gleichnamigen Tabelle anzeigen. Dazu ändern wir den Typ der Entität **Artikel** per Kombinationsfeld auf **Details** und den Typ des Steuerelements **KategorieID** auf **ComboBox**. Dann ziehen Sie die Entität **Artikel** in das neue Fenster **ArtikelMitKombinationsfeld** und sortieren die enthaltenen Steuerelemente so wie in Bild 1 um. Damit die Daten auch im Fenster angezeigt werden, sind in aller Kürze die folgenden Schritte nötig:

- Als Erstes ändern Sie den Entwurf im XAML-Code wie in Listing 1 dargestellt. Hier kommen vor allem die neue **CollectionViewSource** für die Kategorien und die Änderungen am **ComboBox**-Element zum Tragen.

- Dann erweitern Sie den Code des Code behind-Moduls wie in Listing 2. Hier ist die Deklaration der beiden **CollectionViewSource**-Elemente und des Datenbankkontexts sowie deren Füllung im Ereignis **Window_Loaded**, das übrigens auch im XAML-Code noch hinzugefügt werden muss. Details hierzu erfahren Sie im Artikel **EDM: Einfaches Detailfenster**.

Auf die Beschreibung der Navigationsschaltflächen verzichten wir an dieser Stelle – wie dies funktioniert, haben wir ebenfalls im Artikel **EDM: Einfaches Detailfenster** erläutert.

ComboBox zum Hinzufügen von Artikeln?

Nun schauen wir uns an, wie wir dem Kombinationsfeld einen neuen Eintrag unterjubeln können, der dann optimaler-

```
<Window x:Class="BlaetternInDetails.ArtikelMitKombinationsfeld" ...
        Title="ArtikelMitKombinationsfeld" Height="250" Width="300" Loaded="Window_Loaded">
    <Window.Resources>
        <CollectionViewSource x:Key="artikelViewSource"
            d:DesignSource="{d:DesignInstance {x:Type local:Artikel}, CreateList=True}"/>
        <CollectionViewSource x:Key="kategorienViewSource"
            d:DesignSource="{d:DesignInstance {x:Type local:Kategorie}, CreateList=True}"/>
    </Window.Resources>
    <Grid DataContext="{StaticResource artikelViewSource}">
        <Grid x:Name="grid1" HorizontalAlignment="Left" Margin="22,11,0,0" VerticalAlignment="Top">
            ...
            <ComboBox x:Name="kategorieIDComboBox" ItemsSource="{Binding Source={StaticResource kategorienViewSource}}"
                DisplayMemberPath="Bezeichnung" SelectedValuePath="ID" SelectedValue="{Binding Path=KategorieID}">
            </ComboBox>
            ...
        </Grid>
    </Grid>
</Window>
```

Listing 1: Notwendige Änderungen am Fenster zur Anzeige von Artikeln mit Kategorien im Kombinationsfeld

weise direkt in der zugrunde liegenden Tabelle **Kategorien** gespeichert wird. Wenn wir die Anwendung starten und in unserem Beispielfenster versuchen, wie etwa unter Access eine neue Kategorie einzugeben, stellen wir allerdings fest, dass ein **ComboBox**-Steuerelement wohl nicht zum Hinzufügen von Daten gemacht ist (siehe Bild 2). Was nun? Ein anderes Steuerelement wählen? Oder gibt es Einstellungen für das **ComboBox**-Steuerelement, die das Hinzufügen von eigenen Texten ermöglichen? Schauen wir uns doch zunächst eine einfache und auch für den Benutzer intuitive Variante an, bei der wir neben dem Kombinationsfeld eine Schaltfläche zum Hinzufügen eines Eintrags platzieren. Anschließend betrachten wir die direkte Eingabe neuer Elemente direkt in das Kombinationsfeld.

```
public partial class ArtikelMitKombinationsfeld : Window {
    public ArtikelMitKombinationsfeld() {
        InitializeComponent();
    }
    CollectionViewSource artikelViewSource;
    CollectionViewSource kategorienViewSource;
    BestellverwaltungEntities DBContext = new BestellverwaltungEntities();
    private void Window_Loaded(object sender, RoutedEventArgs e) {
        artikelViewSource = ((CollectionViewSource)(this.FindResource("artikelViewSource")));
        DBContext.Artikel.Load();
        artikelViewSource.Source = DBContext.Artikel.Local;
        kategorienViewSource = ((CollectionViewSource)(this.FindResource("kategorienViewSource")));
        DBContext.Kategorien.Load();
        kategorienViewSource.Source = DBContext.Kategorien.Local;
    }
}
```

Listing 2: Anpassung der Code behind-Klasse des Fensters

Bild 2: Das **ComboBox**-Steuerelement erlaubt nur die Auswahl, aber kein Anfügen von Einträgen.

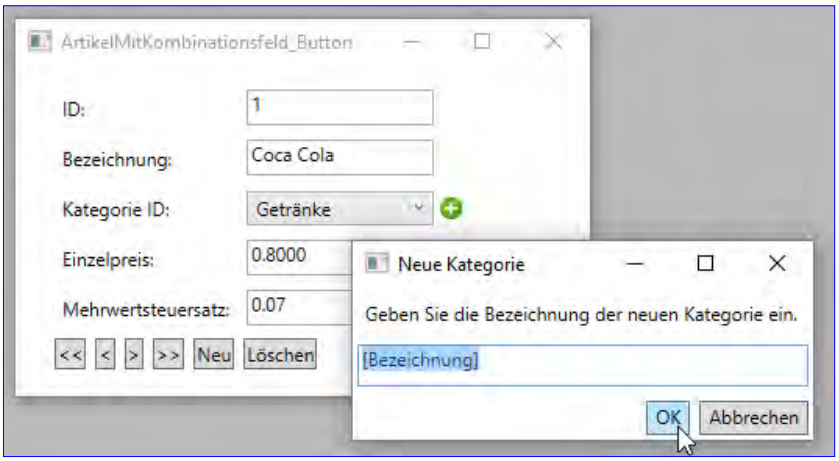


Bild 3: Kombinationsfeld mit Schaltfläche zum Hinzufügen neuer Einträge

Einträge hinzufügen per Schaltfläche

Das Anbieten einer Schaltfläche mit einem Plus-Symbol scheint die offensichtlichere und intuitivere Variante zu sein, um Elemente zu einem Lookup-Kombinationsfeld hinzuzufügen. Nach einem Klick auf diese Schaltfläche soll eine Inputbox erscheinen, die den neuen Eintrag ermittelt (siehe Bild 3). Nach der Eingabe soll die Datenherkunft des Kombinationsfeldes direkt aktualisiert und der neue Eintrag übernommen werden, sodass das Kombinationsfeld diesen direkt anzeigt.

Im Entwurf des Fensters **ArtikelMitKombinationsfeld_Button** sieht dies wie in Bild 4 aus. Hier haben wir neben dem Kombinationsfeld eine neue Schaltfläche hinzugefügt, die wir mit einigen neuen Attributwerten ausgestattet haben.

Die Definition sehen Sie in Listing 3. Wie Sie hier erkennen, haben wir nicht nur einfach eine Schaltlfäche hinzugefügt, sondern diese noch zusammen mit dem **ComboBox**-Element in einem **StackPanel** zusammengefasst. Dieses weist für die Eigenschaft **Orientation** den Wert **Horizontal** auf und soll in der gleichen Grid-Zeile landen wie zuvor das Kombinationsfeld. Zu den Ressourcen des Grids haben wir außerdem noch einen Verweis auf die Datei **add.png** hinzugefügt, welche wir natürlich auch im Projekt gespeichert haben, und mit dem **Key**-Wert **add** versehen.

Die Schaltfläche **btnAdd** verwendet diese Ressource als Wert der **Content**-Eigenschaft (**Content="{StaticResource add}"**). Außerdem haben wir hier den Hintergrund mit der **Background**-Eigenschaft auf **Transparent** eingestellt und den Rahmen mit **BorderThickness="0"** ausgeblendet. Außerdem hinterlegen wir natürlich eine Ereignismethode für den Button, die wie in Listing 4 aussieht.

Die Methode speichert einen Verweis auf das Kombinationsfeld **cboKategorien** in der Variablen **cbo** des Typs **ComboBox**. Dann verwendet sie die im Artikel **Inputbox im Eigenbau** vorgestellte **InputBox**-Klasse, um eine Inputbox zur Eingabe des neuen Eintrags anzuzeigen, und speichert das Ergebnis in der Variablen **strNeu**. Als Standardwert übergeben wir der **Show**-Methode der Klasse dabei den Wert **[Bezeichnung]**. Die folgende **if**-Bedingung prüft dann, ob die Inputbox noch den Wert **[Bezeichnung]** enthält oder

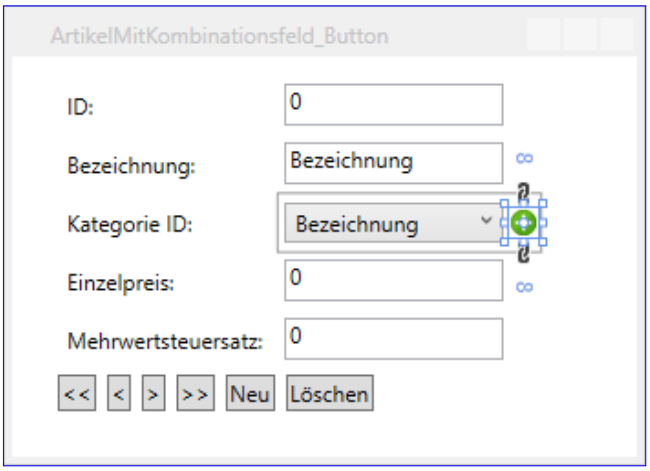


Bild 4: Schaltfläche zum Hinzufügen neuer Einträge

EDM: 1:n-Beziehung als Parent-Child-Ansicht

Eine der unter Access ganz einfach abzubildenden Hierarchien zweier Tabellen ist die 1:n-Beziehung in einem Haupt- und einem Unterformular. Auch unter WPF mit den Daten aus einem Entity Data Model lässt sich dies gut abbilden. Wir schauen uns in diesem Artikel an, wie es funktioniert und was Sie beim Anzeigen der Daten beachten müssen. Als Beispiel nutzen wir die Kategorien der Datenbank im Hauptfenster und die der aktuellen Kategorie zugeordneten Artikel in einer Listenansicht.

Als Beispiele verwenden wir die beispielsweise im Artikel **EDM: Einfaches Detailfenster** vorgestellten Tabellen der Datenbank namens **Bestellverwaltung**, in diesem Fall speziell die Tabelle **Kategorien** für das Hauptfenster, die Tabelle **Artikel** für die Liste sowie die Tabelle **Lieferanten** als Lookuptabelle der Tabelle **Artikel** zur Auswahl der Lieferanten.

Hauptfenster anlegen

Für das Hauptfenster fügen Sie dem Projekt ein neues WPF-Fenster namens **KategorienUndArtikel** hinzu. Nun wollen wir in diesem die Kategorien in der Detailansicht anzeigen, also jeweils nur eine Kategorie gleichzeitig. Darüber hinaus benötigen wir dann natürlich noch Navigationsschaltflächen.

Da wir im Artikel **EDM: Einfaches Detailfenster** ja prinzipiell schon die gleiche Aufgabe erledigt haben, können wir uns dort bedienen (im Beispielprojekt finden Sie auch die in diesem Artikel behandelten Fenster). Als Erstes kümmern wir uns um die zugrunde liegenden Daten. Dazu fügen wir dem XAML-Code unterhalb des **Window**-Elements gleich die Referenz auf die Datenquelle hinzu:

```
<Window.Resources>
    <CollectionViewSource x:Key="vsKategorien"
        d:DesignSource="{d:DesignInstance {x:Type
        local:Kategorie}, CreateList=True}"/>
</Window.Resources>
```

Im Code behind-Modul legen wir die Variable für die **CollectionViewSource** an, diesmal im Gegensatz zu den vom Visual Studio automatisch per Drag and Drop hinzugefügten CollectionViewSources aus den anderen Beispielen mit dem Präfix **vs**, also unter dem Namen **vsKategorien**. Außerdem deklarieren und initialisieren wir auch hier den Datenbankkontext namens **DBContext** auf Basis der Klasse **BestellverwaltungEntities**:

```
CollectionViewSource vsKategorien;
BestellverwaltungEntities DBContext =
    new BestellverwaltungEntities();
```

Dann statten wir die Konstruktor-Methode, die beim Initialisieren des Fensters **KategorienUndArtikel** ausgelöst wird, mit ein paar zusätzlichen Anweisungen aus, die unsere **CollectionViewSource** mit den Daten der Entitätsliste **Kategorien** des **DBContext**-Objekts füllen:

```
public KategorienUndArtikel() {
    InitializeComponent();
    vsKategorien= ((CollectionViewSource)
        (this.FindResource("vsKategorien")));
    DBContext.Kategorien.Load();
    vsKategorien.Source = DBContext.Kategorien.Local;
}
```

Nun folgt die Definition der Steuerelemente im Fenster. Dazu fügen wir dem bereits vorhandenen **Grid**-Element die statische Ressource von oben als **DataContext** hinzu:

```
<Grid DataContext="{StaticResource vsKategorien}">
```

Danach definieren wir für das Grid zwei Spalten und vier Zeilen. Die ersten beiden Spalten sollen die Felder **ID** und **Bezeichnung** der Entität **Kategorie** anzeigen. Dazu stellen

```
<Grid DataContext="{StaticResource vsKategorien}">
    <Grid.ColumnDefinitions>
        <ColumnDefinition Width="Auto"/>
        <ColumnDefinition Width="Auto"/>
    </Grid.ColumnDefinitions>
    <Grid.RowDefinitions>
        <RowDefinition Height="Auto"/>
        <RowDefinition Height="Auto"/>
        <RowDefinition Height="Auto"/>
        <RowDefinition Height="Auto"/>
    </Grid.RowDefinitions>
    <Label Content="ID:" Grid.Column="0" HorizontalAlignment="Left" Margin="3" VerticalAlignment="Center"/>
    <TextBox x:Name="txtID" Grid.Column="1" HorizontalAlignment="Left" Height="22" Margin="3,4,0,6" Text="{Binding ID,
        Mode=TwoWay, NotifyOnValidationError=true, ValidatesOnExceptions=true}" VerticalAlignment="Center" Width="120"/>
    <Label Content="Bezeichnung:" Grid.Column="0" HorizontalAlignment="Left" Margin="3" Grid.Row="1"
        VerticalAlignment="Center"/>
    <TextBox x:Name="txtBezeichnung" Grid.Column="1" HorizontalAlignment="Left" Height="23" Margin="3"
        Grid.Row="1" Text="{Binding Bezeichnung, Mode=TwoWay, NotifyOnValidationError=true, ValidatesOnExceptions=true}"
        VerticalAlignment="Center" Width="120"/>
    ...
</Grid>
```

Listing 1: Definition der Steuerelemente des Hauptfensters

wir das Attribut **Text** auf einen entsprechenden **Binding**-Ausdruck ein, der die Eigenschaft **ID** referenziert. Gleiches erledigen wir für das Textfeld **txtBezeichnung**, welches auf die Eigenschaft **Bezeichnung** zeigt (siehe Listing 1).

Nun fehlen allerdings noch die Navigationsschaltflächen. Diese können Sie einfach in einem Stück aus dem XAML-Code des Fensters **ArtikelMitKombinationsfeld** als Grid in die dritte Zeile des Hauptgrids hineinkopieren. Am XAML-Code brauchen Sie keine Änderungen vorzunehmen. Das Fenster sieht nun im Entwurf wie in Bild 1 aus.

Bild 1: Aktuelle Entwurfsansicht des Fensters

Code für Navigationsschaltflächen kopieren und anpassen

Danach kopieren Sie die Ereignismethoden aus der Code behind-Klasse des Fensters **ArtikelMitKombinationsfeld** in die Code behind-Klasse des neuen Fensters **KategorienUndArtikel**. Die Methoden passen Sie derart an, dass Sie den Namen der **CollectionViewSource** austauschen sowie an allen Stellen die Liste **Artikel** durch die Liste **Kategorien** und die Entität **Artikel** durch die Entität **Kategorie** ersetzen. Das Ergebnis sieht wie in Listing 2 aus. Danach können Sie das Projekt schon starten und die Datensätze des neuen Fensters mit den Navigationsschaltflächen durchlaufen.

```
private void btnErster_Click(object sender, RoutedEventArgs e) {
    Speichern();
    if (vsKategorien.View.CurrentPosition > 0) {
        vsKategorien.View.MoveCurrentToFirst();
    }
}
private void btnVorheriger_Click(object sender, RoutedEventArgs e) {
    Speichern();
    if (vsKategorien.View.CurrentPosition > 0) {
        vsKategorien.View.MoveCurrentToPrevious();
    }
}
private void btnNaechster_Click(object sender, RoutedEventArgs e) {
    Speichern();
    if (vsKategorien.View.CurrentPosition < ((CollectionView)vsKategorien.View).Count) {
        vsKategorien.View.MoveCurrentToNext();
    }
}
private void btnLetzter_Click(object sender, RoutedEventArgs e) {
    Speichern();
    if (vsKategorien.View.CurrentPosition < ((CollectionView)vsKategorien.View).Count) {
        vsKategorien.View.MoveCurrentToLast();
    }
}
private void Speichern() {
    if (DBContext.ChangeTracker.HasChanges()) {
        DBContext.SaveChanges();
    }
}
private void btnNeu_Click(object sender, RoutedEventArgs e) {
    DBContext.Kategorien.Add(DBContext.Kategorien.Create());
    vsKategorien.View.MoveCurrentToLast();
}
private void btnLoeschen_Click(object sender, RoutedEventArgs e) {
    Kategorie kategorie = (Kategorie)vsKategorien.View.CurrentItem;
    if (kategorie != null) {
        DBContext.Kategorien.Remove(kategorie);
        Speichern();
    }
}
```

Listing 2: Ereignismethoden für die Navigationsschaltflächen

Artikel-DataGrid hinzufügen

Nun fehlen aber natürlich noch die Artikel zu der jeweils angezeigten Kategorie. Wie geht man dies an, wenn man keine Anleitung hat? Tendenziell würde man vermutlich davon ausgehen, dass wie etwa für das Füllen eines Kombinationsfeldes eine weitere **CollectionViewSource** nötig ist, um die Daten für die per 1:n-Beziehung verknüpften Daten der Tabelle **Artikel** in einem **DataGrid** oder in einem ähnlichen Steuerelement bereitzustellen. Schauen wir uns also an, wie das aussehen kann.

Synchronisieren per Code

Ausgehend vom obigen Stand (das Ergebnis finden Sie im Fenster **KategorienUndArtikel_II**) fügen wir also zunächst ein weiteres **CollectionViewSource**-Element zum XAML-Code mit dem bereits vorhandenen **CollectionViewSource**-Element hinzu:

```
<Window.Resources>
    <CollectionViewSource x:Key="vsKategorien" />
    <CollectionViewSource x:Key="vsArtikel" />
</Window.Resources>
```

Ein passendes Objekt deklarieren wir auch in der Code behind-Klasse:

```
CollectionViewSource vsArtikel;
```

Dieses füllen wir analog zur CollectionViewSource **vsKategorien** aus dem Entity Data Model:

```
public KategorienUndArtikel_II() {
    InitializeComponent();
    vsKategorien = ((CollectionViewSource)
        (this.FindResource("vsKategorien")));
    DBContext.Kategorien.Load();
    vsKategorien.Source = DBContext.Kategorien.Local;
    vsArtikel = ((CollectionViewSource)
        (this.FindResource("vsArtikel")));
    vsArtikel.Source = DBContext.Artikel.Local;
}
```

Danach fügen wir ein **DataGrid**-Steuerelement zum Fenster hinzu. Dieses soll in der dritten Zeile des vorbereiteten Grid-Steuerelements Platz finden. Damit dieses den übrigen Platz neben den drei bereits gefüllten Zeilen einnimmt, stellen wir die Höhe dieses Elements entsprechend ein:

```
<Grid.RowDefinitions>
    <RowDefinition Height="Auto"/>
    <RowDefinition Height="Auto"/>
    <RowDefinition Height="*"/>
    <RowDefinition Height="Auto"/>
</Grid.RowDefinitions>
```

Die Definition des Grids mit dem DataGrid sieht wie in Listing 3 aus. Wir haben den **DataContext** im übergeordneten **Grid**-Element definiert und auf die CollectionViewSource **vsArtikel** eingestellt. Das DataGrid brauchen wir dann nur noch mit der Eigenschaft **ItemsSource** und dem Wert **{Binding}** an die Datenquelle des übergeordneten Elements zu binden. Die vier **DataGridTextColumn**-Elemente zeigen die Inhalte der vier Eigenschaften **ID**, **Bezeichnung**, **Einzelpreis** und **Kategorien.Bezeichnung** an. Letzteres ist eine Syntax, die für ehemalige Access-Entwickler sehr einfach erscheinen muss: Wir beziehen uns hier einfach auf die Auflistung **Kategorien**, die ja im Entity Data Model definiert ist, und wählen ihre Ei-

AUSGABE 06/2016

DATENBANK ENTWICKLER

MAGAZIN FÜR DIE DATENBANKENTWICKLUNG MIT VISUAL STUDIO FÜR DESKTOP, WEB UND CO.

TOP-THEMEN:

STEUERELEMENTE	Das DockPanel	SEITE 3
WPF-BASICS	WPF-Trigger	SEITE 6
C#-BASICS	PropertyChanged	SEITE 18
DATENZUGRIFF	Validierung mit IDataErrorInfo	SEITE 37
LÖSUNGEN	Kunden verwalten mit Ribbon	SEITE 45

André Minhorst Verlag

WPF-Steuerelemente: Das DockPanel

Das DockPanel-Element ist eines der Steuerelemente, mit dem sich untergeordnete Steuerelemente ausrichten lassen. Das interessante an diesem Steuerelement ist, dass die enthaltenen Elemente jeweils an der angegebenen Seite des DockPanels angedockt werden. Damit lassen sich interessante Anordnungen erzielen. Welche das sind und wie Sie diese realisieren, erfahren Sie in diesem Artikel.

Grundlagen

Neben dem **DockPanel** gibt es noch einige weitere Elemente zum Anordnen der untergeordneten Elemente. Das bekannteste ist wohl das **Grid**-Element, das ja auch standardmäßig als einziges untergeordnetes Element in neuen **Window**- oder **Page**-Elementen angezeigt wird. Während sich mit dem **Grid**-Steuerelement ein Raster, bestehend aus Zeilen und Spalten, realisieren lässt, bietet das **DockPanel** ganz andere Möglichkeiten.

Bild 1: Beispiel für vier an den verschiedenen Seiten angedockte Steuerelemente

Sie können diesem beliebig viele Unterelemente zuweisen, für die Sie jeweils angeben, ob diese oben, unten, links oder rechts innerhalb des DockPanels angedockt werden sollen. Das angegebene Element nimmt dann, wenn es links oder rechts angeordnet wurde, jeweils die gesamte Höhe ein, und wenn es oben oder unten angeordnet wurde, die gesamte Breite.

Die Reihenfolge entscheidet dann darüber, welchem Element welcher Platz zufällt. Wie dies aussieht, zeigen wir anhand von Beispielen. Im ersten Fall haben wir vier **Button**-Elemente angeordnet, die über die Eigenschaft **DockPanel.Dock** mit den Werten **Top**, **Left**, **Bottom** und **Right** versehen wurden (siehe Beispielprojekt unter **Beispiel #1**) – siehe Listing 1.

Bild 1 zeigt, wie das Ergebnis im Fenster aussieht. Das erste Element landet oben und nimmt die komplette Breite ein. Das zweite Element landet links, erhält aber nur noch die Höhe, die das erste Element übrig lässt. Die folgenden Elemente führen dies fort, bis das vierte Element den verbleibenden Platz einnimmt. Das Verhalten des letzten Elements lässt sich übrigens durch das Attribut **LastChildFill** des **DockPanel**-Elements beeinflussen. Stellen Sie dieses auf den Wert **False** ein, nimmt das letzte Element nur den Platz ein, den der Inhalt erfordert (siehe Beispielprojekt unter **Beispiel #2**):

```
<DockPanel>
    <Button DockPanel.Dock="Top" Content="Erstes Element, angedockt an TOP"></Button>
    <Button DockPanel.Dock="Left" Content="Zweites Element, &#xa;angedockt an LEFT"></Button>
    <Button DockPanel.Dock="Bottom" Content="Drittes Element, angedockt an BOTTOM"></Button>
    <Button DockPanel.Dock="Right" Content="Viertes Element, angedockt an RIGHT"></Button>
</DockPanel>
```

Listing 1: Code für vier Elemente, die an den vier Seiten eines Fensters angedockt werden

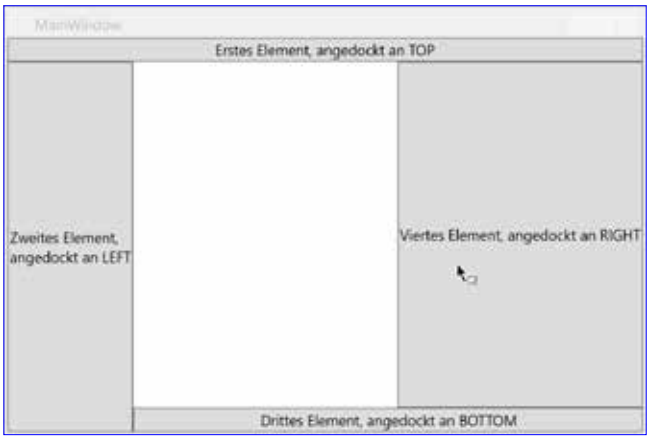


Bild 2: Leerraum durch das Attribut **LastChildFill**

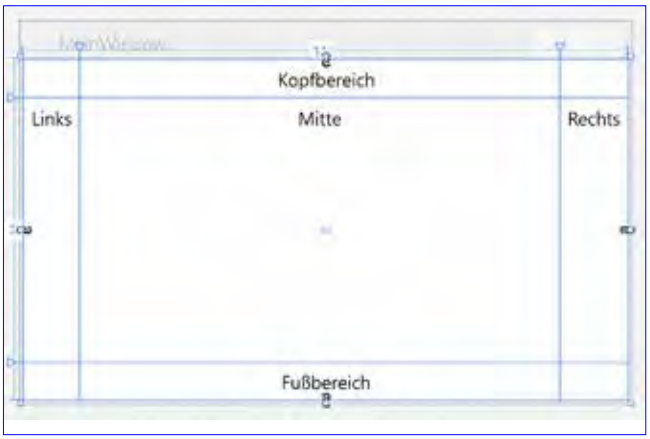


Bild 3: Herstellung eines ähnlichen Layouts mit dem **DockPanel**-Element

```
<DockPanel LastChildFill="False">
    ...
</DockPanel>
```

Dadurch entsteht ein Leerraum wie in Bild 2.

Struktur durch passende Reihenfolge

Vielleicht möchten Sie einmal eine Anordnung erstellen, die zum Beispiel einen Kopfbereich und einen Fußbereich, jeweils über die vollständige Fensterbreite, und dazwischen drei Spalten über die verbleibende Höhe erhalten. Wenn Sie dies mit einem **Grid**-Element erledigen wollen, sieht der Code wie in Listing 2 aus (siehe Beispielprojekt, **Beispiel #3**).

Das Ergebnis erhalten Sie in Bild 3.

Mit dem **DockPanel**-Element lässt sich eine ähnliche Anordnung mit viel weniger XAML-Code herstellen (siehe Listing 3 und im Beispielprojekt unter **Beispiel #4**).

```
<Grid>
    <Grid.RowDefinitions>
        <RowDefinition Height="Auto"></RowDefinition>
        <RowDefinition></RowDefinition>
        <RowDefinition Height="Auto"></RowDefinition>
    </Grid.RowDefinitions>
    <Grid.ColumnDefinitions>
        <ColumnDefinition Width="Auto"></ColumnDefinition>
        <ColumnDefinition></ColumnDefinition>
        <ColumnDefinition Width="Auto"></ColumnDefinition>
    </Grid.ColumnDefinitions>
    <Label Grid.Row="0" Grid.ColumnSpan="3" HorizontalAlignment="Stretch"
        Height="25" HorizontalContentAlignment="Center">Kopfbereich</Label>
    <Label Grid.Row="2" Grid.ColumnSpan="3" HorizontalAlignment="Stretch"
        Height="25" HorizontalContentAlignment="Center">Fußbereich</Label>
    <Label Grid.Row="1" Grid.Column="0" HorizontalAlignment="Center">Links</Label>
    <Label Grid.Row="1" Grid.Column="1" HorizontalAlignment="Center">Mitte</Label>
    <Label Grid.Row="1" Grid.Column="2" HorizontalAlignment="Center">Rechts</Label>
</Grid>
```

Listing 2: Code für die Herstellung einer bestimmten Anordnung per **Grid**-Element

Hier werden also schlicht zunächst ein **Label**-Element an der oberen Seite, eines an der unteren Seite und dann drei Elemente an der linken Seite des **DockPanel**-Elements angedockt. Je nach Anforderung können Sie Anordnungen mit dem **DockPanel**-Element also leichter als mit alternativen Elementen herstellen. Das Ergebnis zeigt Bild 4.

Wenn Sie hier nun noch eine dynamische Aufteilung der Breite der Spalten zwischen dem Kopfbereich und dem Fußbereich erhalten wollen, können Sie das

DockPanel-Element mit dem **Grid**-Element kombinieren – zum Beispiel wie in Listing 4 (siehe Beispielprojekt, **Beispiel #5**).

Hier haben wir ein Label-Element oben angedockt, eines am unteren Rand und ein Grid-Element an der linken Seite (wobei es keine Rolle spielt, wo das letzte Element in diesem Fall angedockt wird – es nimmt immer den Platz zwischen den beiden oben und unten angeordneten Elementen ein).

Zusammenfassung und Ausblick

Dieser Artikel hat das **DockPanel**-Element und seine Vorzüge gegenüber anderen Elementen zur Anordnung von Inhalten erläutert. Sie können damit Elemente an die verschiedenen Seiten des **DockPanels** andocken. Dabei spielen die Reihenfolge sowie die Seite, an der neu hinzugefügte Elemente angedockt werden, eine entscheidende Rolle.

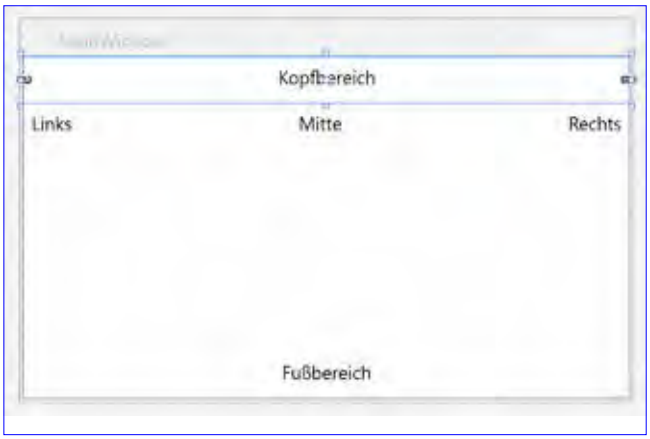


Bild 4: Das gleiche Layout, diesmal mit dem **DockPanel**-Element erstellt

```
<DockPanel>
    <Label DockPanel.Dock="top" Height="30" Content="Kopfbereich" HorizontalAlignment="Stretch"
        HorizontalContentAlignment="Center"></Label>
    <Label DockPanel.Dock="bottom" Height="30" Content="Fußbereich" HorizontalAlignment="Stretch"
        HorizontalContentAlignment="Center"></Label>
    <Label DockPanel.Dock="left" Content="Links"></Label>
    <Label DockPanel.Dock="right" Content="Rechts"></Label>
    <Label DockPanel.Dock="left" Content="Mitte" HorizontalAlignment="Stretch" HorizontalContentAlignment="Center">
        </Label>
</DockPanel>
```

Listing 3: Code, der die gleiche Anordnung per **DockPanel**-Element erzielt

```
<DockPanel>
    <Label DockPanel.Dock="top" Height="30" Content="Kopfbereich"HorizontalAlignment="Stretch"
        HorizontalContentAlignment="Center"></Label>
    <Label DockPanel.Dock="bottom" Content="Fußbereich" HorizontalAlignment="Stretch"
        HorizontalContentAlignment="Center"></Label>
    <Grid DockPanel.Dock="Left">
        <Grid.ColumnDefinitions>
            <ColumnDefinition Width="*"></ColumnDefinition>
            <ColumnDefinition Width="*"></ColumnDefinition>
            <ColumnDefinition Width="*"></ColumnDefinition>
        </Grid.ColumnDefinitions>
        <Label Grid.Column="0" Content="Links" HorizontalAlignment="Center"></Label>
        <Label Grid.Column="1" Content="Mitte" HorizontalAlignment="Center"></Label>
        <Label Grid.Column="2" Content="Rechts" HorizontalAlignment="Center"></Label>
    </Grid>
</DockPanel>
```

Listing 4: Mischung aus **DockPanel** und **Grid**

WPF-Trigger

Trigger kennt der geneigte Access-Anwender höchstens vom SQL Server, wo diese die Möglichkeit boten, auf Änderungen in den Tabellen zu reagieren, für die sie angelegt wurden. Ein Konstrukt namens Trigger gibt es aber auch unter WPF. Dort sind Trigger nicht an Tabellen gebundene Algorithmen, sondern Definitionen von Aktionen, die in Zusammenhang mit Datenänderungen ausgelöst werden. Dieser Artikel stellt Trigger und ihre Anwendungsmöglichkeiten vor.

Trigger-Arten

Es gibt drei Arten von Triggern, von denen zwei aktuell für uns interessant sind. Der Vollständigkeit halber sollen hier jedoch dennoch alle drei Arten aufgeführt werden:

- Eigenschaftstrigger (Property Trigger): Werden ausgelöst, wenn sich der Wert einer Eigenschaft ändert. Arbeitet nur mit Dependency Properties. Kann nur Eigenschaften des betroffenen Elements ändern und dabei nur auf Eigenschaften des gleichen Elements zugreifen.

- Datentrigger (Data Trigger): Wie Eigenschaftstrigger, allerdings auch für andere Eigenschaften als Dependency Properties. Als Quelle kann auch ein anderes Element als das mit dem Trigger versehene Element dienen. Die wichtigste Eigenschaft von Dependency Properties, die hier interessant ist, ist die automatische Information von daran gebundenen Elementen über einen geänderten Zustand des Wertes der Property.

- Ereignistrigger (Event Trigger): Werden zum Beispiel verwendet, um Animationen zu starten und zu beenden, wenn bestimmte Ereignisse eintreten. Um diese Art von Triggern kümmern wir uns in einem späteren Artikel.

Eigenschaftstrigger (Property Trigger)

Gleich zu Beginn zeigen wir Ihnen, dass Sie Trigger ganz einfach statt einfacher C#-Ereignismethoden einsetzen können. Im Beispiel wollen wir die aktive Schaltfläche jeweils mit einem hellgelben Hintergrund versehen. Dazu definieren wir unseren XAML-Code wie folgt (siehe Beispielprojekt, Seite **TriggerPerCSharpEreignis.xaml**):

```
<Grid>
    <Grid.ColumnDefinitions>
        <ColumnDefinition Width="Auto"></ColumnDefinition>
        <ColumnDefinition></ColumnDefinition>
    </Grid.ColumnDefinitions>
    <Grid.RowDefinitions>
        <RowDefinition Height="Auto"></RowDefinition>
        <RowDefinition Height="Auto"></RowDefinition>
    </Grid.RowDefinitions>
    <Label Content="Vorname:" Margin="5"></Label>
    <TextBox x:Name="txtVorname" ... GotFocus="txtVorname_GotFocus" LostFocus="txtVorname_LostFocus"></TextBox>
    <Label Content="Nachname:" Margin="5" Grid.Row="1"></Label>
```

```
    <TextBox x:Name="txtNachname" ... GotFocus="txtNachname_GotFocus" LostFocus="txtNachname_LostFocus"></TextBox>
</Grid>
```

Für die vier für die beiden Textfelder definierten Ereignisattribute hinterlegen wir die folgenden vier Ereignismethoden im Code behind-Modul:

```
private void txtVorname_GotFocus(object sender, RoutedEventArgs e) {
    txtVorname.Background = Brushes.LightYellow;
}
private void txtVorname_LostFocus(object sender, RoutedEventArgs e) {
    txtVorname.Background = null;
}
private void txtNachname_GotFocus(object sender, RoutedEventArgs e) {
    txtNachname.Background = Brushes.LightYellow;
}
private void txtNachname_LostFocus(object sender, RoutedEventArgs e) {
    txtNachname.Background = null;
}
```

Dies sorgt dafür, dass das jeweilige Textfeld bei Fokuserhalt jeweils auf die Hintergrundfarbe **LightYellow** eingestellt wird. Bei Fokusverlust leeren wir die Eigenschaft **Background** wieder, indem wir diese auf den Wert **null** einstellen.

Dies ergibt ein Fenster wie in Bild 1.

Bild 1: Fenster mit zwei Textfeldern, von denen das aktive jeweils einen gelben Hintergrund erhält

Eigenschaftsänderung per XAML

Dies können Sie auch ohne C#-Code erreichen, allerdings mit etwas mehr Schreibaufwand. Ändern Sie den Code für die erste TextBox wie folgt (Beispielprojekt, Seite **PropertyTrigger.xaml**):

```
<TextBox x:Name="txtVorname" Grid.Row="1" Grid.Column="1" Margin="5">
    <TextBox.Style>
        <Style TargetType="{x:Type TextBox}">
            <Style.Triggers>
                <Trigger Property="IsFocused" Value="True">
                    <Setter Property="Background" Value="LightYellow"></Setter>
                </Trigger>
            </Style.Triggers>
        </Style>
    </TextBox.Style>
</TextBox>
```

Abhängige Eigenschaften per Binding

Im Artikel Trigger haben Sie erfahren, dass Sie Eigenschaften von Elementen abhängig von der Änderung anderer Eigenschaften ebenfalls ändern können. Dies gelingt, mitunter über kleine Umwege, auch mithilfe von Bindungen zwischen den Steuerelementen. So können Sie beispielsweise ganz einfach definieren, dass eine Eigenschaft den Wert True erhält, wenn eine Eigenschaft eines anderen Elements auch diese Eigenschaft annimmt. Soll hingegen der Wert False übernommen werden, wenn die andere Eigenschaft True lautet, wird es kompliziert – dann kommt ein Converter ins Spiel. Dieser Artikel zeigt die Möglichkeiten für den Ersatz von Triggern durch Binding auf.

Beispielprojekt

Wir verwenden hier wieder das Beispiel, das auch im Artikel **Trigger** zum Einsatz kam: Wir wollen also in Abhängigkeit vom Zustand eines **CheckBox**-Elements drei Textfelder aktivieren/deaktivieren. Der Aufbau der vier beteiligten Steuerelemente sieht, bevor wir die benötigten Bindungen hinzufügen, wie folgt aus (siehe Bild 1):

Bild 1: Abhängigkeit per Binding realisieren

```
<CheckBox x:Name="chkGeschaeftskunde" Margin="5" Grid.Column="1" VerticalAlignment="Center" />
<TextBox x:Name="txtFirma" Grid.Row="2" Grid.Column="1" Margin="5" />
<TextBox x:Name="txtVorname" Grid.Row="3" Grid.Column="1" Margin="5" />
<TextBox x:Name="txtNachname" Grid.Row="4" Grid.Column="1" Margin="5" />
```

Binding für den gleichen Boolean-Wert

Ganz einfach ist die Lösung, wenn das Ändern einer Eigenschaft eines Elements eine Eigenschaft des anderen Elements mit dem gleichen Datentyp auslösen soll und die Werte der beiden Eigenschaften gleich sein sollen. Das ist bei uns der Fall, wenn wir das **CheckBox**-Element **chkGeschaeftskunde** und das **TextBox**-Element **txtFirma** betrachten: Wenn die Eigenschaft **IsChecked** von **chkGeschäftskunde** den Wert **True** hat, soll auch die Eigenschaft **IsEnabled** von **txtFirma** den Wert **True** aufweisen und umgekehrt. Beide Attribute haben den Datentyp **Boolean**. In diesem Fall können wir den Wert der Eigenschaft **IsChecked** von **chkGeschaeftskunde** direkt per Binding der Eigenschaft **IsEnabled** von **txtFirma** zuweisen:

```
<TextBox x:Name="txtFirma" ... IsEnabled="{Binding ElementName=chkGeschaeftskunde, Path=IsChecked}" />
```

Wann immer der Benutzer nun das Kontrollkästchen mit einem Haken versieht, wird auch das Textfeld aktiviert.

Binding für negierte Boolean-Werte

Bei dem Kontrollkästchen und den beiden übrigen Textfeldern wird es etwas komplizierter: Wir wollen ja nicht den gleichen Wert zuweisen, sondern den negierten Wert – hat die Eigenschaft **IsChecked** von **chkGeschaeftskunde** den Wert **True**, soll **IsEnabled** von **txtVorname** und **txtNachname** den Wert **False** aufweisen und umgekehrt. In einem Binding können wir aber nicht einfach wie unter C# den Operator **Not** verwenden, um einen **Boolean**-Wert umzukehren. Hier benötigen wir einen Converter,

der in Form einer eigenen Klasse mit zwei Methoden daherkommt.

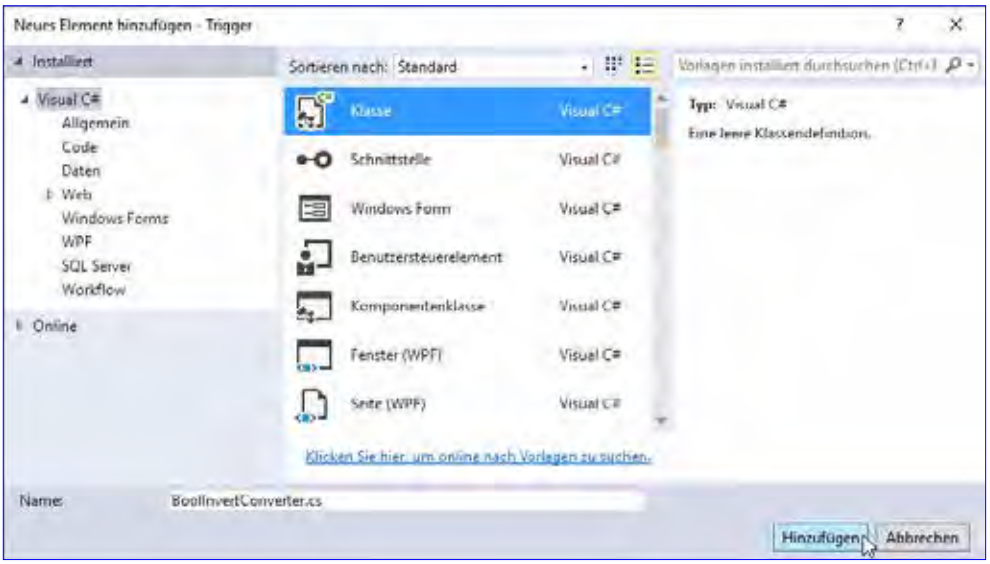


Bild 2: Anlegen der Converter-Klasse

Ein Converter kann einfache Aufgaben übernehmen wie etwa das Umwandeln von **True** in **False** und umgekehrt, aber auch komplexere Operationen durchführen. In diesem Fall wollen wir eine **Converter**-Klasse namens **BoolInvertConverter** erstellen, indem wir per **Strg + Umschalt + A** den Dialog **Neues Element hinzufügen** öffnen, dort den Eintrag **Klasse** auswählen und den Namen **BoolInvertConverter.cs** angeben, bevor wir auf **Hinzufügen** klicken (siehe Bild 2). Nach dem Hinzufügen legen Sie in der Klasse noch einen Verweis auf einen Namespace an, der die gleich verwendete Schnittstelle **IValueConverter** nutzt:

```
using System.Windows.Data;
```

Für die Implentierung eines Converters gibt es bestimmte Regeln, die sich in Form einer Schnittstellendefinition niederschlagen. In der neu erstellten Klasse fügen Sie nun also durch einen Doppelpunkt getrennt hinter dem Klassennamen die Schnittstelle **IValueConverter** hinzu:

```
public class BoolInvertConverter : IValueConverter {
}
```

IValueConverter wird nun rot unterstrichen dargestellt, was daran liegt, dass wir die Member der Schnittstelle noch nicht implementiert haben. Das holen Sie am einfachsten nach, indem Sie mit der rechten Maustaste auf den Namen der Schnittstelle klicken, den Kontextmenü-Eintrag **Schnellaktionen und Refactorings...** auswählen und im nun erscheinenden Popup auf **Schnittstelle implementieren** klicken. Dies fügt die folgenden beiden Methoden zur Klasse hinzu:

```
public object Convert(object value, Type targetType, object parameter, CultureInfo culture) {
    throw new NotImplementedException();
}
public object ConvertBack(object value, Type targetType, object parameter, CultureInfo culture) {
    throw new NotImplementedException();
}
```

Wir benötigen nur die erste Methode **Convert** und ergänzen den Code dieser Methode wie folgt:

Basics: PropertyChanged

Unter Access/VBA haben Sie Tabellen direkt an Formulare und Steuerelemente gebunden, Änderungen an den Daten wurden regelmäßig auch im Frontend aktualisiert. Unter C#/WPF sieht das ganz anders aus: Hier landen die Daten aus der Tabelle erstmal in Objekten und deren Eigenschaften werden mit Steuerelementen wie TextBox, ComboBox und so weiter angezeigt. Damit sich eine Änderung am zugrunde liegenden Objekt auch in der Benutzeroberfläche manifestiert, sind ein paar zusätzliche Handgriffe nötig.

Beispiel: Einfache Kundenklasse

Um uns diese Handgriffe anzusehen, entwerfen wir ein ganz einfaches Beispiel. In diesem legen Sie in einem neuen, leeren Projekt des Typs **Visual C#\WPF-Anwendung** namens **PropertyChanged** eine neue Klasse namens **Kunde.cs** an und füllen das dortige **Namespace**-Element **PropertyChanged** wie folgt:

```
public class Kunde {
    string vorname;
    string nachname;
    public string Vorname {
        get { return vorname; }
        set { vorname = value; }
    }
    public string Nachname {
        get { return nachname; }
        set { nachname = value; }
    }
}
```

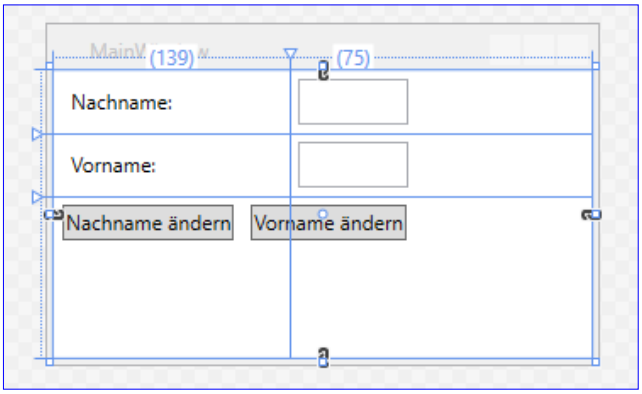


Bild 1: Beispielfenster

Dem Grid des Fensters **MainWindow.xaml** fügen wir in zwei Spalten und drei Zeilen einige Elemente hinzu, die wie folgt definiert werden und die im Entwurf wie in Bild 1 aussehen:

```
<Grid>
    --- Grid-Definition ...
    <Label Content="Vorname:" Grid.Row="1" Margin="5"></Label>
    <TextBox Text="{Binding kunde.Vorname}" Margin="5" Grid.Row="1" Grid.Column="1"></TextBox>
    <Label Content="Nachname:" Grid.Row="0" Margin="5"></Label>
    <TextBox Text="{Binding kunde.Nachname}"  Grid.Row="0" Grid.Column="1" Margin="5"></TextBox>
    <StackPanel Orientation="Horizontal" Grid.Row="2" Grid.ColumnSpan="2">
        <Button x:Name="btnNachnameAendern" Margin="5" Content="Nachname ändern" Click="btnNachnameAendern_Click"></Button>
        <Button x:Name="btnVornameAendern" Margin="5" Content="Vorname ändern" Click="btnVornameAendern_Click"></Button>
    </StackPanel>
</Grid>
```

Die beiden **TextBox**-Elemente sind jeweils über das Attribut **Text** an die Eigenschaften **kunde.Vorname** und **kunde.Nachname** gebunden. Das Objekt **kunde** des Typs **Kunde** enthält, wie aus der Klassendefinition oben ersichtlich, die beiden Eigenschaften

Vorname und **Nachname** und wird beim Erstellen des Fensters **MainWindows** erzeugt. Dafür sorgt die Konstruktor-Methode **MainWindow()**, die beim Erstellen des Objekts ausgelöst wird. Die Klasse enthält außerdem noch eine öffentliche Variable des Typs **Kunde** namens **kunde**, die in der Konstruktor-Methode gefüllt wird:

```
//Code der Klasse MainWindow.xaml.cs
public partial class MainWindow : Window {
    public Kunde kunde { get; set; }          //öffentliche Variable für das anzuzeigende Kunde-Objekt

    public MainWindow() {                     //Konstruktor-Methode
        InitializeComponent();                //Initialisiert das Fenster auf Basis des .xaml-Codes
        kunde = new Kunde();                  //Erstellt ein neues Kunde-Objekt
        kunde.Vorname = "André";              //Füllt die Eigenschaft Vorname
        kunde.Nachname = "Minhorst";          //Füllt die Eigenschaft Nachname
        DataContext = this;                   //Weist die Klasse als Datenquelle für das Fenster zu
    }
...
}
```

Dadurch, dass **DataContext** auf **this** eingestellt ist, können wir im XAML-Code auf alle öffentlichen Elemente dieser Klasse zugreifen. Nach dem Starten sieht die Anwendung nun wie in Bild 2 aus.

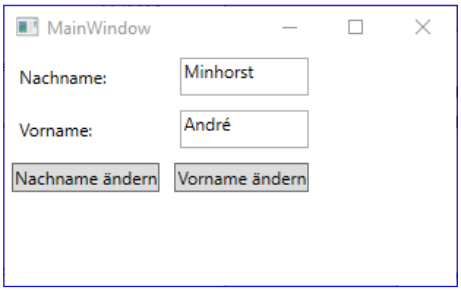


Bild 2: Fenster mit gefüllten Steuerelementen

Eigenschaft aktualisieren

Wozu brauchen wir nun den **PropertyChanged**-Mechanismus? Wir füllen nun die Ereignismethode für die Schaltfläche **btnNachnameAendern** mit folgendem Code:

```
private void btnNachnameAendern_Click(object sender, RoutedEventArgs e) {
    kunde.Nachname = "Neuer Nachname";
    MessageBox.Show("Vorname: '" + kunde.Vorname + "', Nachname: '" + kunde.Nachname + "'", "Neuer Nachname:");
}
```

Dies ändert den Wert der Eigenschaft **Nachname** des angezeigten Objekts auf einen neuen Wert. Außerdem gibt ein Meldungsfenster den neuen Inhalt der beiden Eigenschaften **Vorname** und **Nachname** aus. Für die zweite Schaltfläche hinterlegen wir analog eine Ereignismethode, welche den Wert von **kunde.Vorname** auf einen neuen Wert einstellt und ebenfalls die beiden aktuellen Werte der beiden Eigenschaften ausgibt. Im aktuellen Zustand werden die **MessageBox**-Objekte die neuen Werte anzeigen, aber diese werden nicht in die an die Eigenschaften gebundenen Textfelder übernommen (siehe Bild 3):

```
private void btnVornameAendern_Click(object sender, RoutedEventArgs e) {
    kunde.Vorname = "Neuer Vorname";
    MessageBox.Show("Vorname: '" + kunde.Vorname + "', Nachname: '" + kunde.Nachname + "'", "Neuer Vorname:");
}
```

Events in der Praxis

Im Artikel »Von VBA zu C#: Objekt-Ereignisse« haben wir bereits die Grundlagen zur Programmierung und Implementierung benutzerdefinierter Ereignisse gelegt. Dies wollen wir nun ausbauen, indem wir uns zwei praktische Beispiele ansehen. Dabei wollen wir von einem Hauptfenster aus verschiedene Ansichten in einem Frame anzeigen, darunter eine Kundenübersicht und eine Kundendetailansicht. Beim Anzeigen sollen verschiedene Dinge geschehen, die wir über die Implementierung von Ereignissen lösen wollen – und zwar über eingebaute sowie über benutzerdefinierte Ereignisse.

Beispielprojekt

Im Beispiel geht es um ein Hauptfenster, das über zwei Ribbon-Buttons verschiedene **Page**-Elemente in einem Frame anzeigen soll. Die Schaltfläche **Kundenübersicht** zeigt das **Page**-Objekt **Kundenuebersicht** an, die Schaltlfäche **Neuer Kunde** das **Page**-Objekt **Kundendetails**. Diese Seite wird ebenfalls aufgerufen, wenn der Benutzer doppelt auf einen der Einträge der Kundenübersicht klickt (siehe Bild 1).

Im ersten Beispiel wollen wir dafür sorgen, dass eine dritte Ribbon-Schaltfläche zum Löschen von Elementen der Kundenübersicht nur aktiviert wird, wenn die Kundenübersicht angezeigt wird. Das heißt, dass diese Schaltfläche deaktiviert werden soll, wenn eine andere Ansicht als die Kundenübersicht erscheint. Dazu wollen wir ein Ereignis implementieren, das beim Seitenwechseln im Frame-Element ausgelöst wird.

Im zweiten Beispiel wollen wir zwei eigene Events programmieren und implementieren. Die Kundenübersicht soll nicht jedes Mal, wenn der Benutzer von einer anderen Seite zurück auf diese Seite wechselt, neu erstellt werden beziehungsweise ihre Daten neu aus der Datenbank einlesen. Dies soll nur geschehen, wenn entweder einer der angezeigten Einträge auf der Seite **Kundendetails** geändert oder ein neuer Eintrag hinzugefügt wurde. Dazu wollen wir ein Ereignis in der Code behind-Klasse der Seite **Kundendetails** programmieren, die wir dann in der Code behind-Klasse des **MainWindow**-Objekts implementieren.

So können wir dann direkt im **MainWindow** den relevanten Code ausführen und müssen nicht von der untergeordneten Seite mit den Kundendetails auf das übergeordnete Fenster zugreifen – den Hintergrund erläutern wir weiter unten.

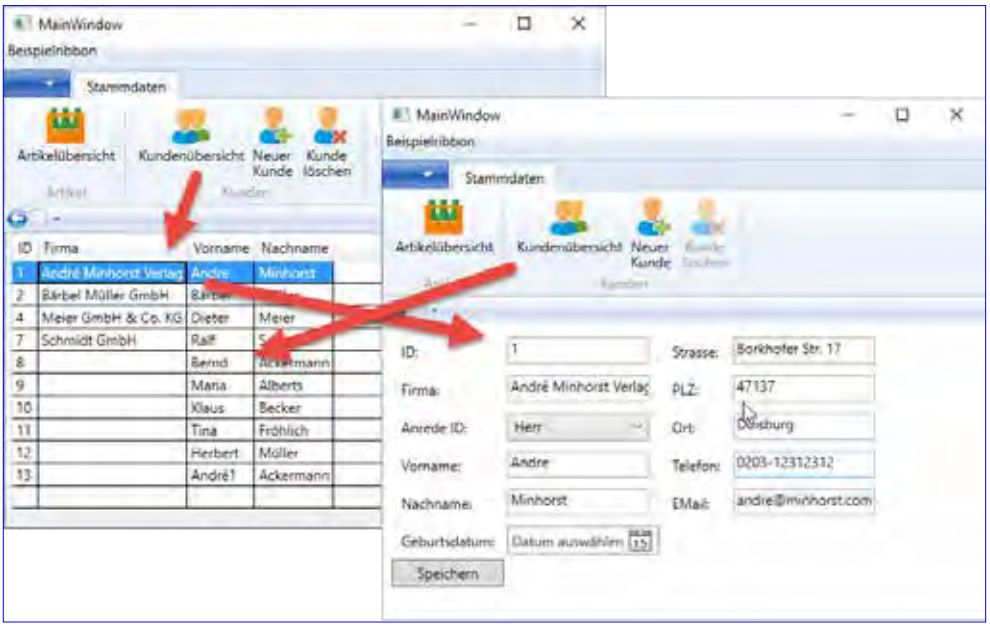


Bild 1: Interaktion zwischen den einzelnen Seiten

Eingebaute Events nutzen

Im ersten Beispiel wollen wir also ein **Ribbon**-Steuerelement im **MainWindow** aktivieren oder deaktivieren, je nachdem welches **Page**-Element das **Frame**-Element anzeigt. Wenn es die Seite **Kundenuebersicht** anzeigt, soll die Schaltfläche **btnKundeLoeschen** aktiviert werden, wenn es eine andere Seite anzeigt, soll der Benutzer diese Schaltfläche nicht anklicken können.

Dazu legen wir für das Frame-Element im XAML-Code, also in der Klasse **MainWindow.xaml**, das Attribut **Navigated** an und füllen es mit dem Wert **WorkZone_Navigated**:

```
<Window x:Class="Bestellverwaltung.MainWindow" ...
        Title="MainWindow" Height="450" Width="525">
    <Grid>
        ...
        <Frame x:Name="WorkZone" Grid.Row="1"
            Navigated="WorkZone_Navigated"></Frame>
    </Grid>
</Window>
```

Für dieses Ereignis hinterlegen wir die folgende Methode in der Code behind-Datei von **MainWindow**:

```
private void WorkZone_Navigated(object sender,
        System.Windows.Navigation.NavigationEventArgs e) {
    switch (WorkZone.Content.ToString()) {
        case "Bestellverwaltung.Kundenuebersicht":
            btnKundeLoeschen.IsEnabled = true;
            break;
        default:
            btnKundeLoeschen.IsEnabled = false;
            break;
    }
}
```

Diese Methode wird nun beim Wechseln des im Frame-Objekts angezeigten Objekts ausgelöst. Mit einer **switch**-Bedingung prüfen wir den Namen des angezeigten Elements, hier mit der Eigenschaft **Content** ermittelt. Hat diese den Wert **Bestellverwaltung.Kundenuebersicht**, stellt die Methode den Wert der Eigenschaft **IsEnabled** des Elements **btnKundeLoeschen** auf **true** ein. Für jeden anderen Inhalt erhält **IsEnabled** den Wert **false**.

Was ist nun der Unterschied zu den Ereignissen etwa einer Schaltfläche, die wir nun schon in vielen Beispielen implementiert haben? In den bisherigen Beispielen haben wir meist selbst die Ereignisse ausgelöst, zum Beispiel durch einen Mausklick auf eine Schaltfläche. Diesmal wurde das Ereignis indirekt durch unsere Aktion ausgelöst, nämlich durch einen Klick auf eine Ribbon-Schaltfläche, die wiederum den Inhalt des **Frame**-Objekts geändert hat. Und das hat dann unser hier implementiertes Ereignis hervorgerufen.

Benutzerdefinierte Events

Wie oben gezeigt, gibt es eine große Anzahl eingebauter Ereignisse, die Sie für Ihre Zwecke nutzen können. Nicht immer jedoch werden Ereignisse zum gewünschten Zeitpunkt ausgelöst. Ein Beispiel ist das Speichern eines Kunden, der neu angelegt oder geändert wurde. Unter Access, wo alles auf die Anzeige und Bearbeitung von Daten ausgelegt ist, finden Sie entsprechende Ereignisse zuhauf – zum Beispiel das Ereignis **Nach Aktualisierung**. Unter WPF/C# sorgen wir selbst dafür, dass die Änderungen an den angezeigten Daten auch in die Datenbank geschrieben werden. Dementsprechend müssen wir auch selbst Ereignisse definieren, die zu einem solchen Zeitpunkt ausgelöst werden.

In unserem Fall wollen wir ein solches Ereignis nutzen, um festzulegen, ob eine Änderung oder eine Neuanlage eines Kunden erfolgt ist. Abhängig davon soll die Kundenübersicht, wenn sie das nächste Mal eingeblendet wird, entweder neu erzeugt werden (falls Änderungen vorliegen) oder mit den alten Daten eingeblendet werden.

Dazu soll das Hauptfenster eine Variable namens **KundenChanged** vorhalten, die lesend und schreibend zugreibar ist und wie folgt in der Klasse **MainWindow** deklariert wird:

```
public bool KundeChanged { get; set; }
```

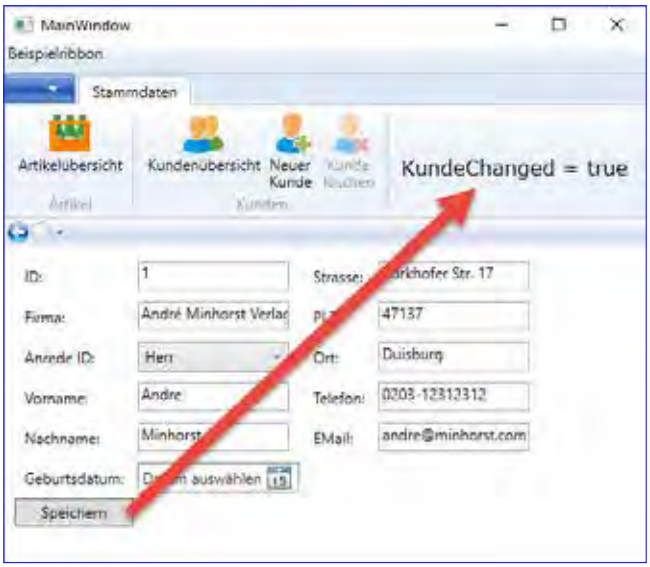


Bild 2: Ein Klick auf **Speichern** stellt die Eigenschaft **KundeChanged** im MainWindow auf **true** ein.

Die einfache Variante

Nun könnten wir im **Page**-Objekt **Kundendetails** die Schaltfläche **Speichern** mit einer Anweisung ausstatten, welche die Eigenschaft **KundeChanged** einfach auf den Wert **true** einstellt, wenn der Benutzer die Schaltfläche betätigt hat (siehe Bild 2).

In diesem einfachen Fall sieht die Methode, die beim Anklicken der **Speichern**-Schaltfläche ausgelöst wird, wie in Listing 1 aus. Die erste Anweisung prüft, ob es sich um einen neuen Kunden handelt (in diesem Fall ist die **ID** noch **0**, da noch nicht vom Datenbanksystem vergeben). Ist der Kunde neu, wird er im ersten Teil der **If**-Bedingung hinzugefügt und in der Datenbank gespeichert. Außerdem referenzieren wir das **MainWindow**-Objekt über **Window.GetWindow(this)** und stellen anschließend seine Eigenschaft **KundeChanged**, die wir weiter oben als öffentliche Variable des Fensters **MainWindow** deklariert haben, auf den Wert **True** ein. Handelt es sich um einen vorhandenen Kunden, der geändert wurde, speichern wir ebenfalls die Änderungen in der Datenbank und stellen dann wie oben den Wert der Eigenschaft **KundeChanged** auf **True** ein.

Neu erstellen oder nicht?

Wenn der Benutzer nun im **MainWindow** auf die Ribbon-Schaltfläche **btnKundenuebersicht** klickt, löst er die Ereignismethode **btnKundenuebersicht_Click** aus Listing 2 aus. Diese prüft im ersten Teil der **if**-Bedingung vor dem Oder-Zeichen (**|**), ob **kundenuebersicht** überhaupt schon existiert, also nicht den Wert **null** enthält – in diesem Fall wird dieses **Page**-Objekt auf jeden Fall neu erstellt. Der Teil hinter dem Oder-Zeichen prüft, ob die Boolean-Variable **KundeChanged** den Wert **true** enthält.

```
private void btnSpeichern_Click(object sender, RoutedEventArgs e) {
    bool kundeCreated = kundeTemp.ID == 0;
    if (kundeCreated) {
        dbContext.Kunden.Add(kundeTemp);
        dbContext.SaveChanges();
        MainWindow wnd =
            (MainWindow)Window.GetWindow(this);
        wnd.KundeChanged = true;
    }
    else {
        dbContext.SaveChanges();
        MainWindow wnd =
            (MainWindow)Window.GetWindow(this);
        wnd.KundeChanged = true;
    }
}
```

Listing 1: Beispiel für das Aktualisieren einer Variablen durch direkten Zugriff

```
private void btnKundenuebersicht_Click(object sender, RoutedEventArgs e) {
    if (kundenuebersicht == null | KundeChanged == true) {
        kundenuebersicht = new Kundenuebersicht();
        KundeChanged = false;
    }
    WorkZone.Content = kundenuebersicht;
}
```

Listing 2: Prüfung, ob Seite neu geladen werden muss

Auch dies kann ein Indikator sein, dass die Seite **kundenuebersicht** neu geladen werden muss, da in diesem Fall ein neuer Datensatz angelegt oder ein bestehender Datensatz geändert wurde. Wird die Seite nun neu erstellt und später über die **Content**-Eigenschaft in das **Frame**-Objekt **WorkZone** geladen, stellt die Methode auch gleichzeitig die Variable **KundeChanged** wieder auf den Wert **false** ein, damit nun nicht bei jedem Aufruf ein Neuerstellen der Seite erfolgt. Perfekt – warum lassen wir es nicht einfach so?

Weil wir so zu viele Abhängigkeiten zwischen den einzelnen Elementen der Benutzeroberfläche schaffen. Wenn die Seite zum Anzeigen der Kundendetails wiederverwendbar sein soll, also wenn wir die Seite auch in anderen Fenstern oder Zusammenhängen anzeigen wollen, dann ist es wichtig, dass die Seite nicht von bestimmten Eigenschaften der übergeordneten Instanz – hier des Fensters mit dem Frame-Element – abhängt. Und genau das geschieht, wenn wir im MainWindow eine öffentliche Eigenschaft anbieten, die von einem untergeordneten Element wie der Seite bei bestimmten Ereignissen geändert oder auch abgefragt werden soll.

Welche Alternativen haben wir also? Wir wollen dafür sorgen, dass das MainWindow davon erfährt, wenn in dem im Frame angezeigten **Page**-Element etwas geschieht, ohne vom untergeordneten Element direkt auf die Eigenschaften des übergeordneten Elements zuzugreifen. Die Lösung ist, in dem untergeordneten Element ein Ereignis zu definieren (oder auch ein vorhandenes Ereignis zu nutzen),

Für Quereinsteiger: Wie es unter Access/VBA läuft

Die Syntax zum Deklarieren öffentlicher Ereignisse sah für diejenigen, die bisher mit Access/VBA gearbeitet haben, recht einfach aus – zum Beispiel so:

```
Public Event Ereignisname(Parameter1 As <Datentyp>, Parameter2 As <Datentyp>, ...)
```

Um das Ereignis auszulösen, war schlicht der Aufruf dieses Ereignisses erforderlich – zum Beispiel wie folgt innerhalb einer Prozedur:

```
Private Sub cmdOK_Click
    Call Ereignisname("Wert1", "Wert2", ...)
End Sub
```

Die Klasse hat man dann in einer anderen Klasse mit dem Schlüsselwort **WithEvents** deklariert:

```
Dim WithEvents objAusloeser As clsAusloeser
```

Danach braucht man nur noch den Steuerelementnamen und den Ereignisnamen aus den beiden Kombinationsfeldern im VBA-Editor auszuwählen, um die Implementierung des Ereignisses wie folgt zu erzeugen:

```
Private Sub objAusloeser_Ereignisname(Parameter 1 As _
        <Datentyp>, Parameter2 As <Datentyp>, ...)
    ...
End Sub
```

Unter C# sieht das Ganze wesentlich komplizierter aus, eröffnet dadurch aber natürlich auch mehr Flexibilität.

Der Ablauf

Wenn wir zwei benutzerdefinierte Ereignisse definieren, die durch das Anlegen eines neuen Kunden beziehungsweise das Ändern eines vorhandenen Kunden ausgelöst werden, geschieht erst einmal nichts bei diesen beiden Aktionen. Die Ereignisse werden ausgelöst, aber wenn diese nicht implementiert werden, passiert auch nichts. Erst wenn wir wie in diesem Beispiel in **MainWindow.xaml.cs** Ereignismethoden definieren, welche die Ereignisse **ItemChanged** oder **ItemAdded** implementieren, können wir die Ereignisse nutzen. Den allgemeinen Ablauf zeigt Bild 3.

Die Ereignisse werden dann in **Kundendetails.xaml.cs** ausgelöst, was die Ereignismethoden in **MainWindow.xaml.cs** auf den Plan ruft. Deren Implementierung stellt schließlich die Eigenschaft **KundeChanged** auf **true** ein, sodass die Kundenübersicht beim nächsten Anzeigen neu erstellt wird. Wir schauen uns nun zwei verschiedene Varianten zum Definieren und Nutzen von benutzerdefinierten Events an.

EDM: Ausnahmen beim Speichern behandeln

Die Validierung bei der Eingabe von Daten ist eines der wichtigsten Themen bei der Erstellung benutzerfreundlicher Anwendungen. In diesem ersten Artikel zu diesem Thema wollen wir uns darauf beschränken, solche Eingabefehler abzufangen, welche durch die Restriktionen im Datenmodell und entsprechende Fehleingaben entstehen. Das sind beispielsweise Fehler, die auftreten, weil der Benutzer keinen Wert in ein Feld eingibt, das nicht leer sein darf, oder der Datentyp des Feldes nicht mit dem eingegebenen Wert korrespondiert.

Beispielanwendung

Wie bei den meisten Artikeln dieser Ausgabe wollen wir unser Know-how an der Beispielanwendung **Bestellverwaltung** ausprobieren. Diese hat in einigen Tabellen Restriktionen, gegen die wir über die Eingabe in den beiden vorgestellten Seiten **Kundenuebersicht** oder **Artikeluebersicht** verstoßen können. Das wären beispielsweise Folgende:

- Das Feld **Bezeichnung** in der Tabelle **Artikel** darf nicht **Null** sein, muss also einen Wert enthalten.

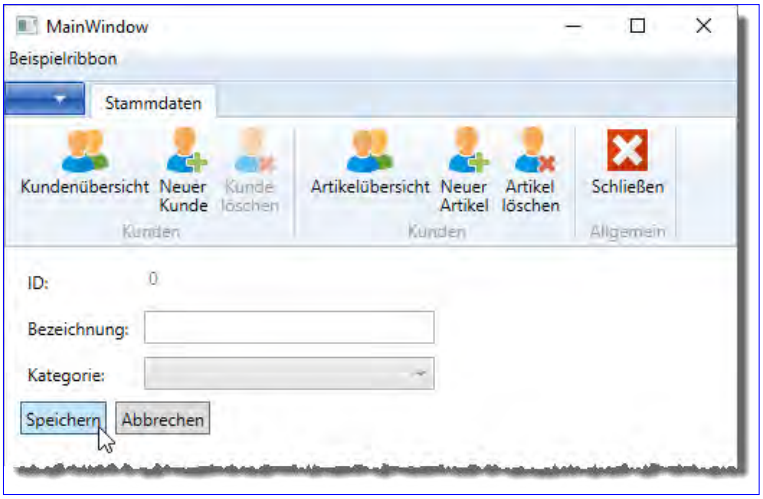


Bild 1: Speichern eines Datensatzes ohne Angabe der Bezeichnung

- Für die Fremdschlüsselfelder **AnredeID** der Tabelle **Kunden** oder **KategorieID** der Tabelle **Artikel** muss ein Wert angegeben werden.

- Andere Felder sind grundsätzlich aufgrund ihres Datentyps auf die Eingabe bestimmter Werte beschränkt – zum Beispiel können Sie keinen Text in ein Zahlen- oder Datumsfeld eingeben.

Ablauf ohne Validierung

Schauen wir uns an, was geschieht, wenn wir keine Vorsichtsmaßnahmen treffen und davon ausgehen, dass der Benutzer die korrekten Werte in die Felder eingibt und auch alle Pflichtfelder ausfüllt. Nehmen wir also an, dass der Benutzer einen neuen Artikel anlegt und vor dem Klick auf die Schaltfläche **Speichern** keine Bezeichnung eingibt und auch keine Kategorie auswählt (siehe Bild 1).

Dann liefert Visual Studio die Ausnahme aus Bild 2. Die hier angegebene Information, es sei eine Ausnahme des Typs **System.Data.Entity.Validation.DbEntityValidationException** liefert noch keine echte Hilfe beim Beheben des Problems.

Also klicken Sie auf den Link **Details anzeigen...** und erhalten eine weitere Meldung, die diesmal den Hinweis **Fehler bei der Überprüfung einer oder mehrerer Entitäten** liefert. Wir erfahren noch, dass die Methode **SaveChanges** den Fehler ausgelöst hat, aber das war es dann auch.

Wir können an dieser Stelle nur die Meldung ausblenden und die Ausführung beenden. Selbst wenn Sie den Link **Ausnahmedetail in die Zwischenablage kopieren** anklicken und den Inhalt der Zwischenablage anschließend in einen Editor Ihrer Wahl einfügen, erhalten Sie hier keine

nennenswerten weiteren Informationen.

Einzelnen Fehler provozieren

Schauen wir uns nun an, was geschieht, wenn wir nur die **Bezeichnung** auslassen. In diesem Fall erhalten wir den gleichen Fehler, als wenn wir die Bezeichnung und die Kategorie weglassen. Unter Messsage erhalten wir dann, wenn wir in die Tiefe gehen, die folgende Meldung:

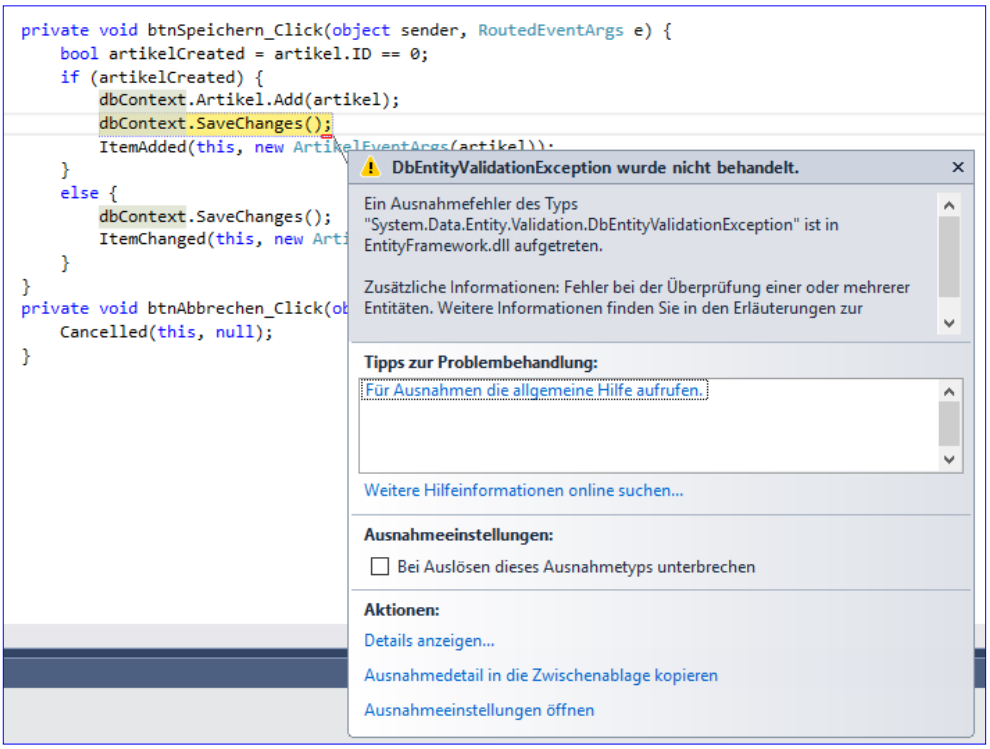


Bild 2: Ausnahme beim Versuch, Daten ohne Füllen eines Feldes zu speichern, das keine Nullwerte akzeptiert

```
Fehler bei der Überprüfung einer oder mehrerer Entitäten. Weitere Informationen finden Sie in den Erläuterungen zur EntityValidationErrors-Eigenschaft.
```

Um den Fehler weiter zu untersuchen, fassen wir die **SaveChanges**-Methode in einen **Try...Catch**-Block ein, der wie folgt aussieht:

```
try {
    dbContext.SaveChanges();
}
catch (Exception ex) {
    message = string.Format("Fehler: {0}", ex.Message);
    Debug.WriteLine(message);
    message = string.Format("Typ: {0}", ex.GetType());
    Debug.WriteLine(message);
}
```

Wir fangen also eine einfache Exception ab und geben im Ausgabefenster den Inhalt der Eigenschaften **Message** und **GetType** des **Exception**-Objekts aus. Das Ergebnis sieht wie folgt aus:

```
Fehler: Fehler bei der Überprüfung einer oder mehrerer Entitäten. Weitere Informationen finden Sie in den Erläuterungen zur EntityValidationErrors-Eigenschaft.
Typ: System.Data.Entity.Validation.DbEntityValidationException
```

Da wir nun wissen, dass es sich um eine Exception des Typs **DbEntityValidationException** handelt, können wir nun gezielt diese Exception abfangen.

DbEntityValidationException abfangen

Die Abfrage der Exception **DbEntityValidationException** bauen wir wie in Listing 1 in die Methode **btnSpeichern_Click** ein. Den bestehenden **Catch**-Zweig mit der Standardexception behalten wir bei, damit auch andere Fehler behandelt werden. Der neue **Catch**-Zweig gibt zunächst eine Meldung aus, dass ein Fehler aufgetreten ist. Dann durchläuft er in einer **foreach**-Schleife alle Elemente des Typs **DbEntityValidationResult** der Auflistung **EntityValidationErrors** und speichert diese in der Variablen **item**.

```
try {
    dbContext.SaveChanges();
}
catch (DbEntityValidationException ex) {
    Debug.WriteLine("Fehler beim Speichern:");
    Debug.WriteLine("=====================================");
    foreach (DbEntityValidationResult item in ex.EntityValidationErrors) {
        DbEntityEntry entry = item.Entry;
        string entityTypeName = entry.Entity.GetType().Name;
        foreach (DbValidationError subItem in item.ValidationErrors) {
            string message = string.Format("Fehlerbeschreibung: '{0}'", subItem.ErrorMessage);
            Debug.WriteLine(message);
            message = string.Format("Tabelle/Entität: {0}", entityTypeName);
            Debug.WriteLine(message);
            message = string.Format("Feld/Eigenschaft: {0}", subItem.PropertyName);
            Debug.WriteLine(message);
        }
    }
    Debug.WriteLine("=====================================");
}
```

Listing 1: Analysieren eines Fehlers

Die Variable **entry** mit dem Typ **DbEntityEntry** erhält den Inhalt der **Entry**-Eigenschaft des in **item** gespeicherten Elements. Der Zugriff auf die Eigenschaft **Name** der Funktion **GetType()** des **Entity**-Objekts aus dem **DBEntityEntry**-Element liefert den Namen der Entität, hier **Artikel**. Da für das Speichern eines Objekts dieser Entität in der Datenbank mehrere Fehler gleichzeitig aufgetreten sein können, durchlaufen wir die Auflistung **ValidationErrors** des **DBEntityEntry**-Objekts aus **item** – und dies wiederum in einer **foreach**-Schleife. Diese liefert mit den beiden Eigenschaften **ErrorMessage** und **PropertyName** zwei wichtige Informationen, nämlich die Fehlermeldung und die Eigenschaft, deren Speichern den Fehler ausgelöst hat. Das Ausgabe-Fenster zeigt die gewünschten Informationen nun wie in Bild 3 an. Wir wissen nun also, dass die Eingabe eines Wertes für das Feld **Bezeichnung** erforderlich ist und könnten mit den Informationen, die hier geliefert wurden, auch eine entsprechende Validierungsmeldung ausgeben.

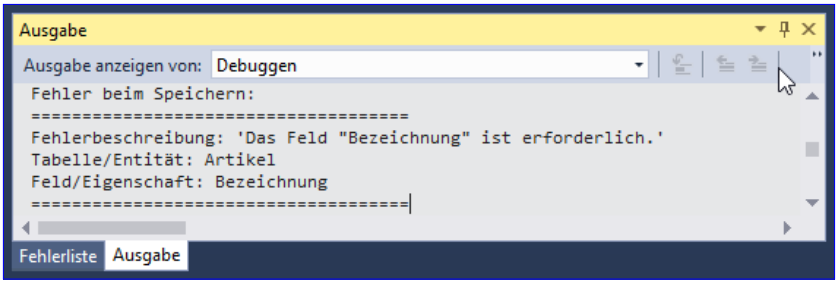


Bild 3: Ausgabe der Informationen bei der **DBEntityValidationException**

Fehlender Fremdschlüsselwert

Schauen wir uns nun an, ob wir auch das Problem mit dem Fremdschlüsselfeld auf diese Weise in den Griff bekommen. Wenn Sie also erneut einen neuen Artikel anlegen wollen und diesmal zwar die Bezeichnung eingeben, aber keinen Wert für das Fremdschlüsselfeld **KategorieID** auswählen, greift wieder die allgemeine Exception im letzten **Case**-Zweig. Dies gibt die folgende Meldung aus, die ohne Exception-Behandlung wie in Bild 4 aussehen würde:

```
Fehler: Fehler beim Aktualisieren der
Einträge. Weitere Informationen finden
Sie in der internen Ausnahme.
Typ: System.Data.Entity.Infrastructure.
DbUpdateException
```

EDM: Validieren von Entitäten mit IDataErrorInfo

Die Validierung bei der Eingabe von Daten ist eines der wichtigsten Themen bei der Erstellung benutzerfreundlicher Anwendungen. Nachdem der Artikel »EDM: Ausnahmen beim Speichern behandeln« gezeigt hat, wozu Sie im Rahmen der Validierung die durch Restriktionen im Datenmodell auftretenden Exceptions nutzen können, schauen wir uns nun einen einfachen Weg an, um Validierungsregeln in Entitätsklassen zu definieren und beim Fehlschlagen der Validierung entsprechende Meldungen in der Benutzeroberfläche auszugeben. Dabei zeigen wir hier den Umgang mit der Schnittstelle »IDataErrorInfo«.

Beispielanwendung

Wie im oben genannten Artikel nutzen wir wieder die Beispielanwendung **Bestellverwaltung**. Bevor die im Datenmodell dieser Anwendung festgelegten Restriktionen greifen und bei Fehleingaben Exceptions auslösen, wollen wir Validierungen programmieren, um Fehleingaben des Benutzers zu vermeiden. In diesem Fall schauen wir uns die Seite **Kundendetails.xaml.cs** an, um die Programmierung der Validierung zu veranschaulichen. Dies soll im Ergebnis wie in Bild 1 aussehen.

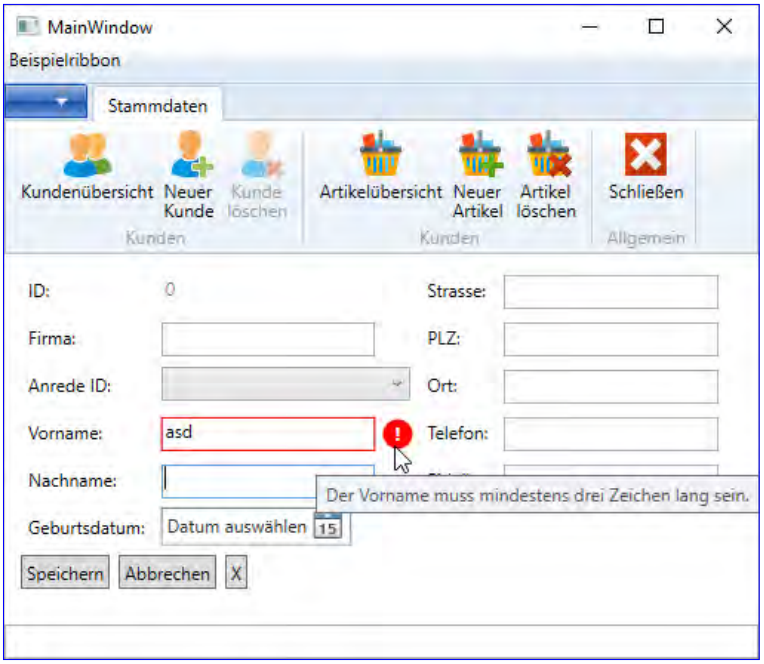


Bild 1: Beispiel für eine einfache Validierungsmeldung

Validierung

Die hier vorgestellten Techniken zur Validierung von Benutzereingaben beziehen sich auf die Untersuchung der jeweiligen Entität – ist ein Wert in einem Feld mit einem eindeutigen Index bereits vorhanden? Darf ein Feld überhaupt leer sein? Hat der Benutzer auch ein gültiges Datum eingegeben? Liegt der Zahlenwert im zulässigen Bereich?

Validierung auf Feldebene

Validierungen können Sie auf mehrere Arten durchführen. Der Artikel **EDM: Ausnahmen beim Speichern behandeln** zeigt, wie Sie erst beim Auftreten von Fehlern beim Speichern auf fehlerhafte Eingaben des Benutzers reagieren können. Davor gibt es noch mindestens zwei weitere Methoden:

- Validieren direkt nach der Eingabe eines Wertes in ein Steuerelement und Hinweis auf die fehlerhafte Eingabe
- Validieren, nachdem der Benutzer den Speichervorgang initiiert hat, aber bevor das Speichern selbst stattfindet. Hier können dann nicht nur einzelne Felder isoliert, sondern auch mehrere Felder im Zusammenhang untersucht werden. Wie dies gelingt, zeigt zum Beispiel der Artikel **EDM: Validieren von Entiten mit IValidatableObject**.

In diesem Artikel schauen wir uns die einfachere Validierung direkt nach der Eingabe eines Wertes an.

Für den Benutzer hat dies beispielsweise den Vorteil, dass er Fehleingaben direkt erkennt und diese direkt korrigieren kann.

Einfache Validierung

In vielen Fällen reicht eine einfache Validierung der Eingaben für Steuerelemente aus. Damit meinen wir, dass nur die Werte einzelner Steuerelemente geprüft werden sollen, und zwar direkt nach der Eingabe. So können Sie beispielsweise abfragen, ob ein Wert eine bestimmte Mindestlänge erreicht, und bei Nichterfüllung dieser Vorgabe einen optischen Hinweis einblenden. Dieser Artikel zeigt, wie Sie eine solche Validierung zu einer Anwendung hinzufügen können.

Beispielanwendung

Die Beispielanwendung heißt **Bestellverwaltung**. Die im Fenster **MainWindow** eingeblendete Seite **Kundendetails.xaml** soll zeigen, wie Sie die hier vorgestellte Art der Validierung implementieren.

Ablauf der Validierung

Die Validierung mit der hier vorgestellten Methode geschieht immer gleich nach dem Abschluss der Eingabe in ein Feld. Sie löst dann die Anzeige des Steuerelements mit der fehlerhaften Eingabe mit rotem Rahmen und mit einem Warnsymbol aus. Das Überfahren des Warnsymbols mit der Maus führt zur Anzeige eines ToolTip-Textes, der weitere Informationen zum vorliegenden Fehler liefert.

Aufbau der Validierung

Die Validierung basiert auf zwei verschiedenen Elementen. Das erste Element ist die Klasse, welche die Entität der zu validierenden Objekte beschreibt, in unserem Fall also etwa **Kunde.cs**. Hier legen wir in Form einer Methode die Regeln fest, nach denen das soeben geänderte Steuerelement geprüft werden soll. Damit erhalten wir eine Basis, die wir für alle Zugriffe auf Objekte dieser Klasse nutzen können, was die Wiederverwendbarkeit dieser Klasse erhöht. Der zweite Teil der Validierung ist das Hinzufügen einer bestimmten Eigenschaft zum Binding des Steuerelements, im Falle eines Textfeldes also beispielsweise zum Attribut **Text**. Außerdem müssen wir noch definieren, wie sich das Aussehen des Textfeldes ändern soll, wenn die Validierung fehlgeschlagen ist. Immerhin wollen wir den Benutzer ja auch informieren, wenn dieser eine Fehleingabe tätigt.

Dies erledigen wir durch eine entsprechende Vorlage für das **TextBox**-Steuerelement. Schauen wir uns nun die drei Elemente der Validierung an.

Anpassen der Klasse Kunde.cs

Unser Entity Data Model, das wir auf Basis der SQL Server-Datenbank **Bestellverwaltung** erstellt haben, enthält auch eine Klasse namens **Kunde.cs**, welche die Eigenschaften eines Kunden abbildet. Diese erweitern wir nun um die Schnittstelle **IDataErrorInfo**. Da diese in der Bibliothek **System.ComponentModel** beschrieben wird, fügen wir zunächst einen Verweis auf diese Schnittstelle per **using**-Schlüsselwort hinzu:

```
using System.ComponentModel;
```

Danach legen wir in der Kopfzeile der Klasse fest, dass diese die Schnittstelle **IDataErrorInfo** implementieren soll. Die Zeile sieht dann so aus:

```
public partial class Kunde : IDataErrorInfo {
...
```

Um die Member dieser Schnittstelle schnell zu implementieren, wählen Sie den Kontextmenü-Eintrag **Schnellaktionen und Refactorings** von **IDataErrorInfo** aus. Im nun erscheinenden Popup wählen Sie **Schnittstelle implementieren** aus und erstellen so die beiden Methoden dieser Schnittstelle. Diese sehen zunächst wie folgt aus:

```
public string Error {
    get {
        throw new NotImplementedException();
    }
}

public string this[string columnName] {
    get {
        throw new NotImplementedException();
    }
}
```

Hier fügen wir nun für die Methode **this** eine erste Validierung ein. Diese soll sicherstellen, dass die Eingabe in das Feld **Vorname** nicht offen bleibt.

```
public string this[string columnName] {
    get {
        string errorMessage = "";
        switch (columnName) {
            case "Vorname":
                {
                    if (String.IsNullOrEmpty(Vorname)) {
                        errorMessage = "Bitte geben Sie einen Vornamen ein.";
                    }
                    break;
                }
        }
        return errorMessage;
    }
}
```

Listing 1: Implementierung der Schnittstelle **IDataErrorInfo** in der Klasse **Kunde.cs**

Die Methode **this** nimmt mit dem Parameter **columnName** immer den Namen des Feldes entgegen, das untersucht werden soll – in diesem Fall **Vorname**. Im **get**-Teil der Methode deklarieren wir die Variable **result** als **string** und prüfen in einer **if**-Bedingung, ob **columnName** den Wert **Vorname** enthält. Falls ja, prüfen wir, ob **Vorname** den Wert **null** enthält oder leer ist (**IsNullOrEmpty**). Ist das der Fall, stellen wir den Rückgabewert mit der Variablen **errorMessage** auf den Text **Bitte geben Sie einen Vornamen ein.** ein (siehe Listing 1).

.xaml-Code erweitern

Wenn Sie das Projekt nun starten, einen neuen Kunden anlegen und einen Vornamen mit maximal drei Zeichen angeben, tut sich jedoch erst einmal nichts. Das ist auch kein Wunder, denn wir haben ja im **.xaml**-Code auch noch nicht angegeben, dass es eine Validierung gibt.

Damit die Methode **this** der Klasse **Kunde.cs** überhaupt aufgerufen wird, müssen wir der Bindung des Textfeldes **Vorname** das Attribut **ValidatesOnDataErrors** hinzufügen und dafür den Wert **true** festlegen:

```
<TextBox x:Name="txtVorname" Grid.Column="1" Grid.Row="3"
HorizontalAlignment="Stretch"
Text="{Binding kunde.Vorname, Mode=TwoWay,
ValidatesOnDataErrors=True}" />
```

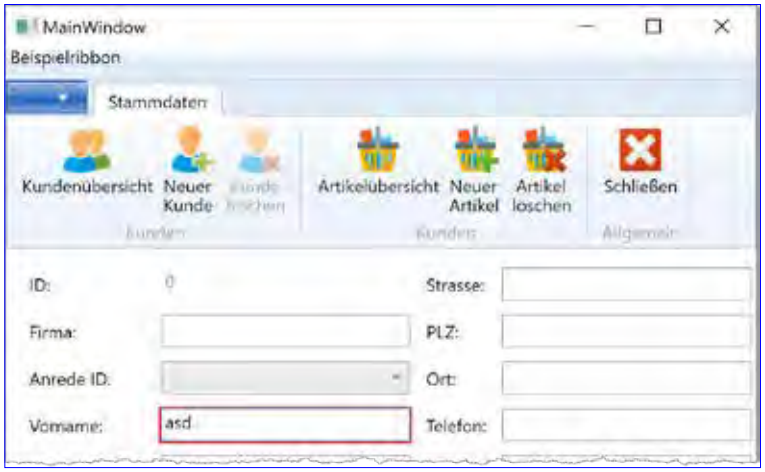


Bild 2: Eine fehlerhafte Eingabe wird durch einen roten Rahmen markiert.

Ein erneuter Start der Anwendung und die Eingabe eines ungültigen Ausdrucks zeigt nun Wirkung: Das fehlerhafte Feld wird mit einem roten Rahmen versehen (siehe Bild 2).

Validierungshinweis anbringen

Nun fehlt noch der Hinweis für den Benutzer, damit er weiß, was er bei der Eingabe falsch gemacht hat. Dazu wollen wir rechts vom Textfeld ein Warnsymbol anzeigen, das beim Überfahren mit der Maus einen ToolTip-Text einblendet. Dazu ist nicht mehr nötig als die Änderung einiger Eigenschaften des **TextBox**-Elements. Da wir diese nicht nur für eine **TextBox** benötigen, sondern für alle, definieren wir direkt

ein entsprechendes **Style**-Element in den Ressourcen des übergeordneten Elements, hier also für ein **Page**-Element.

Also fügen Sie, soweit noch nicht vorhanden, direkt unterhalb des **Page**-Elements ein **Page.Resources**-Element wie das aus Listing 2 hinzu (wenn Sie ein **Window** zur Anzeige der zu validierenden Daten nutzen, handelt es sich um das Element **Window.Resources**).

Das **Style**-Element wird für jeden Elementtyp einmal angelegt und enthält alle Definitionen, die alle Steuerelemente dieses Typs betreffen. In diesem Fall legen wir diese mit dem Attribut **TargetType** für den Typ **TextBox** fest.
Die ersten drei **Setter**-Elemente definieren die Höhe, den linken, oberen, rechten und unteren Abstand des eigentlichen Elements zu dem dafür reservierten Bereich und die vertikale Ausrichtung.

Das vierte **Setter**-Element liefert den Inhalt für die Eigenschaft **Validation.ErrorTemplate**. Da es sich hier nicht um einen einfachen Attribut-Wert handelt, sondern um eine komplette Struktur, legen wir diese Struktur in Form von Unterelementen unterhalb des Elements **Setter.Value** an. Dass die darunter angegebenen Elemente nur bei einem Validierungsfehler im gebundenen Feld des betroffenen Steuerelements angezeigt werden, liegt am Namen der Property, also **Validation.ErrorTemplate**.

Damit unser Warnsymbol, übrigens ein einfacher roter Kreis mit einem Ausrufezeichen in der Mitte, sichtbar wird, fügen wir zunächst ein **DockPanel**-Element hinzu. Darin bringen wir ein **Border**-Element unter, das im **DockPanel** rechts verankert wird und die Höhe und Breite 20 hat. Mit dem Wert 10 für das Attribut **CornerRadius** wird das bisherige, 20x20 große Viereck zu einem Kreis. Diesem verleihen wir noch

```
<Page.Resources>
    ...
    <Style TargetType="{x:Type TextBox}">
        <Setter Property="Height" Value="23" />
        <Setter Property="Margin" Value="3,3,27,0" />
        <Setter Property="VerticalAlignment" Value="Center" />
        <Setter Property="Validation.ErrorTemplate">
            <Setter.Value>
                <ControlTemplate>
                    <DockPanel>
                        <Border Background="Red" DockPanel.Dock="right" Margin="5,0,0,0" Width="20" Height="20" CornerRadius="10"
                ToolTip="{Binding ElementName=customAdorner, Path=AdornedElement.(Validation.Errors)[0].ErrorContent}">
                            <TextBlock Text="!" VerticalAlignment="center" HorizontalAlignment="center"
                                FontWeight="Bold" Foreground="white" />
                        </Border>
                        <AdornedElementPlaceholder Name="customAdorner" VerticalAlignment="Center" >
                            <Border BorderBrush="red" BorderThickness="1" />
                        </AdornedElementPlaceholder>
                    </DockPanel>
                </ControlTemplate>
            </Setter.Value>
        </Setter>
    </Style>
</Page.Resources>
```

Listing 2: Hinzufügen der Elemente zur Anzeige eines Warnsymbols und eines ToolTip-Textes

EDM: Kunden verwalten mit Ribbon

Bisher haben wir in diesem Magazin nur einzelne Beispiele zur Darstellung von Daten aus Tabellen geliefert. Diesmal wollen wir einen Schritt weiter gehen: Wir erstellen eine WPF-Anwendung, die ein eigenes Ribbon enthält und mit diesem die Steuerung einiger Funktionen zur Auflistung von Kunden sowie zum Bearbeiten, Hinzufügen und Löschen von Kundendatensätzen ermöglicht. Dabei nutzen wir als Container für die angezeigten Seiten mit der Kundenliste und den Kundendetails ein Frame-Objekt. Damit können wir, wenn mehrere Benutzer geöffnet sind, sogar durch die entsprechenden Seiten navigieren.

Dabei wollen wir zunächst eine Kundenübersicht und später eine Seite zum Anlegen eines neuen Kunden über entsprechende Ribbon-Steuerelemente sichtbar machen. Eine weitere Schaltfläche soll den aktuell in der Kundenübersicht markierten Kunden löschen.

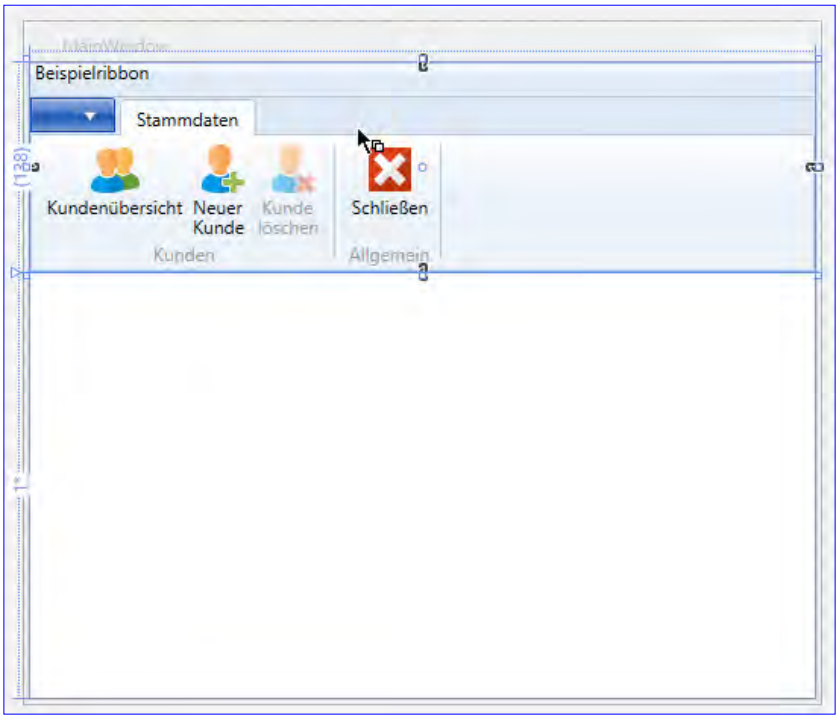


Bild 1: Erster Entwurf des Ribbons

Das heißt, dass wir zunächst ein Ribbon im Fenster **MainWindow** unserer Beispielanwendung benötigen, das drei Schaltflächen für die gewünschte Navigation in den Kundendatensätzen und eine Schaltfläche zum Schließen des Fensters enthält. Dies sollte später so wie in Bild 1 aussehen.

Um dies zu realisieren, legen Sie zunächst ein neues Projekt des Typs **Visual C#|WPF-Anwendung** namens **NavigationMitRibbons** an. Um die Ribbons zu definieren, benötigen Sie einen neuen Verweis, den Sie über den **Verweis-Manager** (Menü-Eintrag **Projekt|Verweise**) hinzufügen. Der Verweis heißt **System.Windows.Controls.Ribbons**.

Da wir dem **Window**-Element im Kopf ein Ribbon und darunter die Steuerelemente zur Anzeige der Artikel- und der Kundenübersicht hinzufügen wollen, legen wir zunächst ein Grid mit zwei Zeilen an. Diese definieren wir wie folgt:

```
<Grid.RowDefinitions>
    <RowDefinition Height="Auto"></RowDefinition>
    <RowDefinition></RowDefinition>
</Grid.RowDefinitions>
```

Anschließend fügen wir dem Grid ein **Ribbon**-Element mit zwei Schaltflächen in verschiedenen Gruppen hinzu, das wir über das Attribut **Grid.Row="0"** der ersten Ribbonzeile zuweisen (siehe Listing 1). Diesem fügen wir ein **RibbonTab**-Element mit zwei **RibbonGroup**-Elementen hinzu. Die

```
<Ribbon Name="rbnMain" Title="Beispielribbon" Grid.Row="0">
    <RibbonTab Name="Tab1" Header="Stammdaten" KeyTip="A">
        <RibbonGroup Header="Kunden">
            <RibbonButton Name="btnKundenuebersicht" Label="Kundenübersicht" Click="btnKundenuebersicht_Click"
                LargeImageSource="images\users.png"></RibbonButton>
            <RibbonButton Name="btnNeuerKunde" Label="Neuer Kunde" Click="btnNeuerKunde_Click"
                LargeImageSource="images\user_add.png"></RibbonButton>
            <RibbonButton Name="btnKundeLoeschen" Label="Kunde löschen" Click="btnKundeLoeschen_Click"
                LargeImageSource="images\user_delete.png" IsEnabled="false"></RibbonButton>
        </RibbonGroup>
        <RibbonGroup Header="Allgemein">
            <RibbonButton Name="btnSchliessen" Label="Schließen" Click="btnSchliessen_Click"
                LargeImageSource="images\close.png"></RibbonButton>
        </RibbonGroup>
    </RibbonTab>
</Ribbon>
```

Listing 1: Code für ein Ribbon mit Schaltflächen in verschiedenen Gruppen

erste Gruppe statten wir mit drei **RibbonButton**-Elementen aus und die zweite mit einem **RibbonButton**-Element. Die Schaltflächen versehen wir mit Attributen, die auf die jeweiligen Ereignismethoden verweisen. Außerdem legen wir mit der Eigenschaft **LargeImageSource** den Namen jeweils einer Bilddatei fest, die im Ribbon für die jeweilige Schaltfläche angezeigt werden soll. Diese Bilder legen wir in einem Unterordner namens **images** im Projektmappen-Explorer ab.

Seiten erstellen

Nun wollen wir die beiden Seiten erstellen, die durch einen Mausklick auf die beiden Ribbon-Einträge angezeigt werden sollen. Dazu legen Sie zunächst ein erstes neues **Page**-Element an und nennen es **Kundenuebersicht** (**Strg + Umschalt + A**, dann **Seite (WPF)** auswählen und den Namen unten angeben) – siehe Bild 2.

Das Ergebnis ist ein transparentes, rechteckiges Objekt, für das weder Höhe noch Breite mit den üblichen Attributen **Height** oder **Width** festgelegt sind. Stattdessen finden wir die beiden Attribute **d:DesignHeight** und **d:DesignWidth**. Aber was sind das für Eigenschaften? Eigenschaften, die mit **d:** beginnen, sind Eigenschaften für die Design-Ansicht, die nur die Größe des **Page**-Elements für die Entwurfsansicht markieren.

Zur Laufzeit wird das **Page**-Element ja ohnehin in das **Frame**-Element eingebettet und nimmt dessen Größe an. Dementsprechend finden Sie im **Page**-Element die beiden Eigenschaften **d:DesignHeight** und **d:DesignWidth**. Das Attribut

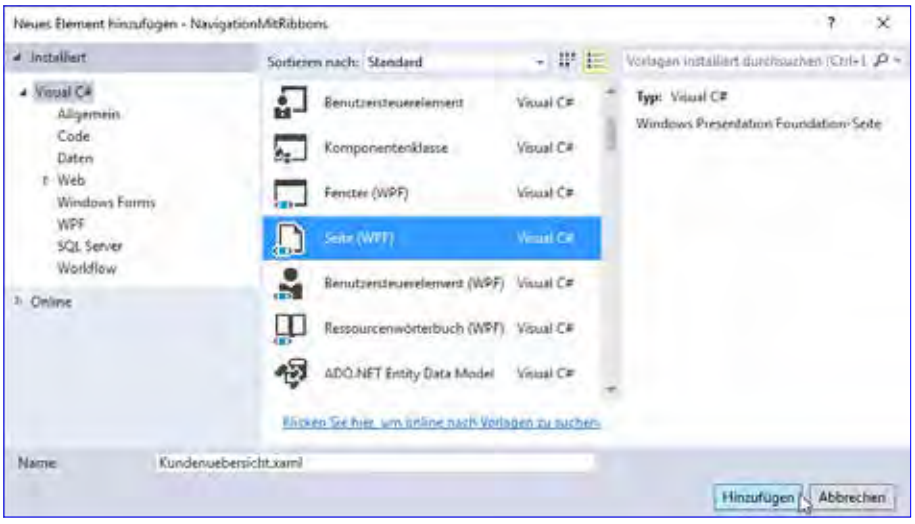


Bild 2: Hinzufügen eines **Page**-Elements

```
public partial class Kundenuebersicht : Page {
    BestellverwaltungEntities dbContext;
    private List<Kunde> kunden;
    public List<Kunde> Kunden {
        get {
            return kunden;
        }
        set {
            kunden = value;
        }
    }
    public Kundenuebersicht(int kundeID = 0) {
        InitializeComponent();
        dbContext = new BestellverwaltungEntities();
        kunden = new List<Kunde>(dbContext.Kunden);
        DataContext = this;
        if (kundeID != 0) {
            Kunde currentKunde = dbContext.Kunden.Find(kundeID);
            dgKunden.SelectedItem = currentKunde;
            dgKunden.ScrollIntoView(currentKunde);
        }
    }
}
```

Listing 2: Code behind-Datei der **Page**-Klasse **Kundenuebersicht.xaml**

mc:Ignorable="d" gibt dem Interpreter zu verstehen, dass Attribute mit führendem **d** nicht interpretiert werden sollen:

```
<Page x:Class="NavigationMitRibbons.Kundenuebersicht"
    ...
    mc:Ignorable="d"
    d:DesignHeight="300" d:DesignWidth="300"
    Title="Kundenuebersicht">
```

Datenquelle

Als Datenquelle für dieses Beispiel verwenden wir wieder die Datenbank **Bestellverwaltung**, die Sie bereits in weiteren Artikeln in diesem Magazin kennen gelernt haben. Dazu haben wir wieder ein Entity Data Model namens **BestellverwaltungEntities** angelegt.

Page mit Kunden füllen

Damit das **Page**-Element die Kunden in einer entsprechenden Liste, hier in Form eines **DataGrid**-Elements, anzeigt, stellen wir zunächst eine entsprechende Datenquelle in Form eines **List**-Objekts in der Code behind-Datei **Kundenuebersicht.xaml.cs** zur Verfügung, die wir innerhalb der Konstruktor-Methode der Klasse füllen (siehe Listing 2).

In der Konstruktor-Methode der Klasse **Kundenuebersicht** wird zunächst die Methode **InitializeComponent** ausgeführt, um das Fenster entsprechend der XAML-Definition aufzubauen. Danach erstellen wir das Entity Data Model-Objekt **dbContext** auf Basis der Klasse **BestellverwaltungEntities**. Die Liste **kunden**, die wir zuvor als privates **List**-Objekt deklariert haben, füllen wir dann mit der Auflistung **Kunden** des **dbContext**-Objekts:

```
kunden = new List<Kunde>(dbContext.Kunden);
```

Danach weisen wir dem **Page**-Objekt mit der **DataContext**-Eigenschaft als Datenherkunft die Code behind-Klasse selbst zu. Damit die Seite nun auf die Kunden-Liste aus der Variablen **kunde** zugreifen kann, stellen wir diese noch mit einer öffentlichen Variablen namens **Kunden** zur Verfügung:

```
public List<Kunde> Kunden {
    ...
}
```

Desweiteren haben Sie sicher bemerkt, dass wir einen optionalen Parameter namens **kundeID** für die Konstruktormethode angelegt haben. Diese wird in der **if**-Bedingung interessant: Ist **kundeID** nämlich nicht **0**, was geschieht, wann immer eine Kunden-ID übermittelt wird, dann liest die Methode den Kunden mit dieser ID in die Variable **currentKunde** ein. Dieser wird dann im DataGrid als aktueller Kunde

```
<Page x:Class="Bestellverwaltung.Kundenuebersicht" ...  xmlns:local="clr-namespace:Bestellverwaltung"
      mc:Ignorable="d" d:DesignHeight="300" d:DesignWidth="300" Title="Kundenuebersicht">
    <Grid>
        <DataGrid x:Name="dgKunden" ItemsSource="{Binding Kunden}" AutoGenerateColumns="false">
            <DataGrid.Columns>
                <DataGridTextColumn Binding="{Binding Path=ID}" Header="ID"  />
                <DataGridTextColumn Binding="{Binding Path=Firma}" Header="Firma" />
                <DataGridTextColumn Binding="{Binding Path=Vorname}" Header="Vorname"  />
                <DataGridTextColumn Binding="{Binding Path=Nachname}" Header="Nachname" />
            </DataGrid.Columns>
        </DataGrid>
    </Grid>
</Page>
```

Listing 3: Code für die Definition des **Page**-Elements zur Anzeige der Kundenübersicht

markiert. Die Methode **ScrollInto-View** sorgt dann noch dafür, dass dieser Datensatz auch noch in den sichtbaren Bereich verschoben wird. Letzteres ist für Access/VBA-Programmierer natürlich ein Traum – dort waren für eine solche Aktion durchaus größere Verrenkungen nötig.

DataGrid-Element in Page anlegen

Schließlich wollen wir ein **DataGrid**-Element im **Page**-Element anlegen, das wie in Listing 3 definiert wird. Die Seite hat ja als Datenherkunft bereits die Code behind-Klasse erhalten. Damit brauchen wir für das **DataGrid**-Element namens **dgKunden** nur noch das Attribut **ItemsSource** auf den Wert **{Binding Kunden}** einzustellen.

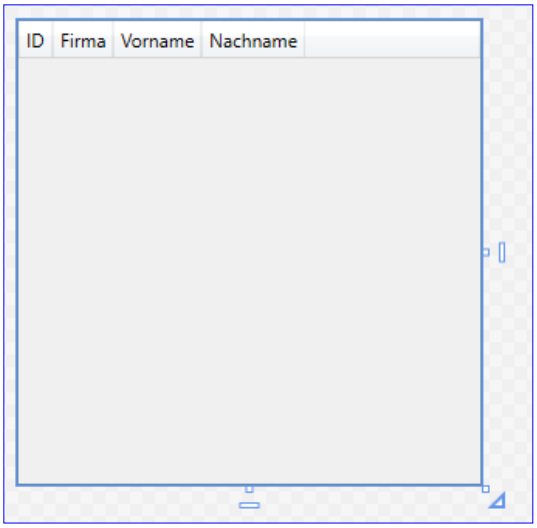


Bild 3: Das **DataGrid** im **Page**-Element in der Entwurfsansicht

Es verwendet dann die Auflistung **Kunden** aus der Klasse **Kunden-uebersicht.xaml.cs** als Datenquelle. Die Spalten sollen nicht automatisch erstellt werden, daher erhält **AutoGenerateColumns** den Wert **false**. Die Spalten legen wir unterhalb des Elements **DataGrid.Columns** an, und zwar als **DataGridTextColumn**-Elemente. Hier erhalten diese per **Binding** eine Verknüpfung zum jeweiligen Feld der Datenherkunft sowie per **Header** eine Spaltenüberschrift. Das Ergebnis soll anschließend wie in Bild 4 aussehen.

```
<Window x:Class="NavigationMitRibbons.MainWindow" ... xmlns:local="clr-namespace:NavigationMitRibbons"
        mc:Ignorable="d" Title="MainWindow" Height="350" Width="525">
    <Grid>
        <Grid.RowDefinitions>...</Grid.RowDefinitions>
        <Ribbon Name="rbnMain" Title="Beispielribbon" Grid.Row="0">...</Ribbon>
        <Frame x:Name="WorkZone" Grid.Row="1"></Frame>
    </Grid>
</Window>
```

Listing 4: Gekürzte Fassung der Definition von **MainWindow.xaml**, jetzt mit dem **Frame**-Objekt

EDM: Kundendetails verwalten

In unserer Lösung zur Verwaltung von Bestellungen spielt die Kundenverwaltung natürlich eine große Rolle. In diesem Artikel wollen wir uns ansehen, wie wir einen neuen Kunden anlegen oder einen vorhandenen Kunden bearbeiten können – eingebettet natürlich in unser Hauptfenster, das wir im Artikel »Kunden verwalten« beleuchten. Daher benötigen wir auch kein eigenes Fenster, um die Kundendetails anzuzeigen, sondern erstellen ein Page-Objekt mit den relevanten Informationen. Dieses nutzen wir dann sowohl zum Anlegen neuer Kunden als auch zum Bearbeiten vorhandener Kunden.

Die Seite zum Bearbeiten und Erstellen von Kunden soll in Aktion wie in Bild 1 aussehen. Wenn der Benutzer auf den Ribbon-Eintrag **Kundenübersicht** klickt, soll die Kundenliste erscheinen, über die per Doppelklick einer der Kunden in der hier gezeigten Detailansicht erscheinen soll. Wenn der Benutzer auf die Schaltfläche **Neuer Kunde** klickt, erscheint die gleiche Seite, allerdings mit einem leeren **Kunden**-Objekt.

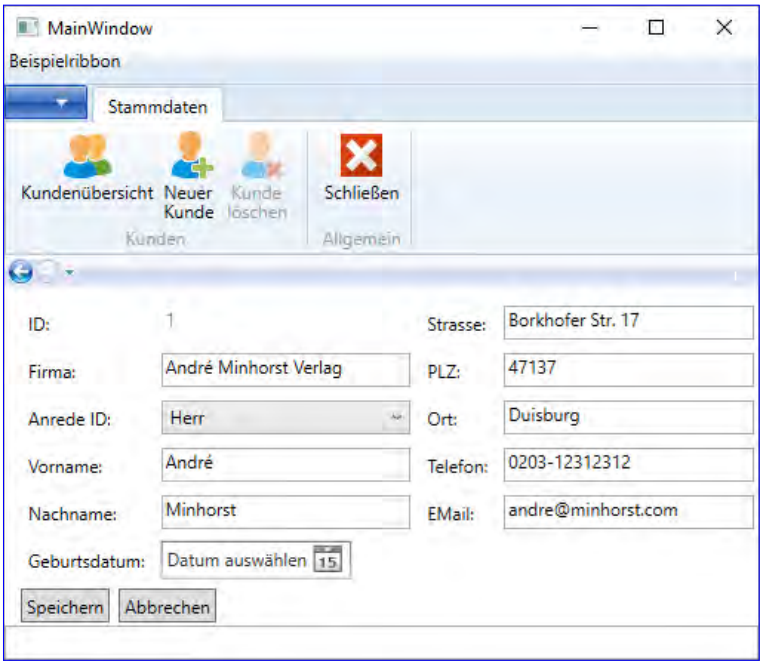


Bild 1: Die Seite zum Bearbeiten und Anlegen von Kunden

Kundendetails-Seite anlegen

Um die Kunden zu bearbeiten, legen Sie in unserem Projekt **Bestellverwaltung** ein Objekt des Typs **Seite** an. Wir wollen diese Seite in ein **Frame**-Element einblenden, wann immer die Bearbeitung eines Kunden nötig ist – oder wenn ein neuer Kunde angelegt werden soll. Erstmal müssen wir die Seite jedoch erstellen und mit den entsprechenden Steuerelementen sowie dem Code zum Füllen der Steuerelemente mit den Daten ausstatten. Die Seite soll im Entwurf wie in Bild 2 aussehen. Die zweite und die vierte Spalte haben wir so ausgelegt, dass sie sich beim Verbreitern des **Page**-Elements ebenfalls im gleichen Maße verbreitern – allerdings nur bis zu einer bestimmten maximalen Breite.

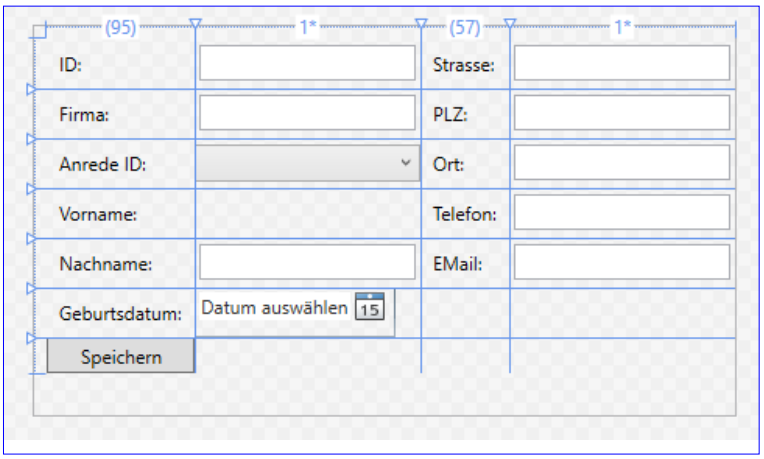


Bild 2: Die Seite zum Bearbeiten und Anlegen von Kunden im Entwurf

Standardwerte für Steuerelementeigenschaften

Den WPF-Code für diese Seite finden Sie in Listing 1. Zu Beginn des Codes haben wir im Breich **Page.**

```
<Page x:Class="Bestellverwaltung.Kundendetails"
      ...
      d:DesignHeight="250" d:DesignWidth="450"
      Title="Kundendetails">
    <Page.Resources>
        <Style TargetType="Label">
            <Setter Property="HorizontalAlignment" Value="Left"></Setter>
            <Setter Property="Margin" Value="3"></Setter>
            <Setter Property="VerticalAlignment" Value="Center"></Setter>
        </Style>
        <Style TargetType="TextBox">
            <Setter Property="Height" Value="23"></Setter>
            <Setter Property="Margin" Value="3"></Setter>
            <Setter Property="VerticalAlignment" Value="Center"></Setter>
        </Style>
    </Page.Resources>
    <Grid x:Name="grid1" HorizontalAlignment="Stretch" Margin="8,8,0,0" VerticalAlignment="Top">
        <Grid.ColumnDefinitions>
            <ColumnDefinition Width="Auto" />
            <ColumnDefinition Width="*" MaxWidth="300"/>
            <ColumnDefinition Width="Auto"/>
            <ColumnDefinition Width="*" MaxWidth="300"/>
        </Grid.ColumnDefinitions>
        <Grid.RowDefinitions>
            <RowDefinition Height="Auto"/>
            ...
        </Grid.RowDefinitions>
        <Label Content="ID:" Grid.Column="0" />
        <TextBox x:Name="txtID" Grid.Column="1" HorizontalAlignment="Left" Text="{Binding kunde.ID, Mode=TwoWay,
            NotifyOnValidationError=true, ValidatesOnExceptions=true}" Width="50" IsEnabled="False" BorderBrush="Transparent"/>
        <Label Content="Firma:" Grid.Column="0" Grid.Row="1" />
        <TextBox x:Name="txtFirma" ... Text="{Binding kunde.Firma, ...}" />
        <Label Content="Anrede ID:" Grid.Column="0" Grid.Row="2" />
        <ComboBox x:Name="cboAnreden" Grid.Column="1" HorizontalAlignment="Stretch" Height="23" Grid.Row="2"
            ItemsSource="{Binding Anreden}" SelectedItem="{Binding kunde.Anreden}" DisplayMemberPath="Bezeichnung"
            SelectedValuePath="ID" VerticalAlignment="Center" Margin="3"></ComboBox>
        <Label Content="Vorname:" ... /><TextBox x:Name="txtVorname" ... Text="{Binding kunde.Vorname, ...}" />
        <Label Content="Nachname:" ... /><TextBox x:Name="txtNachname" ... Text="{Binding kunde.Nachname, ...}" />
        <Label Content="Geburtsdatum:" .../>
        <DatePicker x:Name="dpGeburtsdatum" ... SelectedDate="{Binding kunde.Geburtsdatum, ...}" Margin="3"/>
        <Label Content="Strasse:" ... /><TextBox x:Name="txtStrasse" ... Text="{Binding kunde.Strasse, ...}" />
        <Label Content="PLZ:" ... /><TextBox x:Name="txtPLZ" Text="{Binding kunde.PLZ, ...}" />
        <Label Content="Ort:" ... /> <TextBox x:Name="txtOrt" ... Text="{Binding kunde.Ort, ...}" />
        <Label Content="Telefon:" ... /><TextBox x:Name="txtTelefon" ... Text="{Binding kunde.Telefon, ...}" />
        <Label Content="EMail:" ... /><TextBox x:Name="txtEMail" ... Text="{Binding kunde.EMail, ...}" />
        <Button x:Name="btnSpeichern" Click="btnSpeichern_Click" Grid.Row="6" Grid.Column="0" Height="23"
            Content="Speichern"></Button>
    </Grid>
</Page>
```

Listing 1: WPF-Code für die Seite **Kundendetails**

Resources einige **Style**-Elemente hinzugefügt. **Style**-Elemente dienen der Festlegung von Werten für bestimmte Attribute von WPF-Elementen. Das erste verwendet für das Attribut **TargetType** den Wert **Label**, das heißt, dass es Standardwerte für **Label**-Elemente vorgibt. In diesem Fall legen wir mit den untergeordneten Elementen des Typs **Setter** jeweils die Eigenschaft (**Property**) und den Wert (**Value**) der Voreinstellung fest. Das Attribut **HorizontalAlignment** soll den Standardwert **Left** erhalten, **Margin** den Wert **3** und **VerticalAlignment** den Wert **Center**. Diese Werte brauchen wir dann nicht mehr explizit für die Steuerelemente des Typs **Label** festzulegen. Sollten wir an einer bestimmten Stelle einmal andere Werte benötigen, tragen Sie diese einfach für das entsprechende **Label**-Element ein.

Auch für das **TextBox**-Steuerelement haben wir drei Attribute vordefiniert, nämlich **Height** (Wert: **23**), **Margin** (**3**) und **VerticalAlignment** (**Center**).

Damit erhalten wir nicht nur den Vorteil, diese Werte nicht mehr explizit einstellen zu müssen. Wir brauchen außerdem auch Änderungen, die alle Elemente des gleichen Typs betreffen, nur noch an einer Stelle vorzunehmen.

Das Grid

Unser Grid soll vier Spalten und sieben Zeilen enthalten. Für die Spalten 1 und 3 haben wir die Breite **Auto** festgelegt, für die Spalten 2 und 4 den Wert ***** sowie die maximale Breite von **300**.

Die einzelnen Steuerelemente weisen wir den Zellen des Grids zu, indem wir die Eigenschaften **Grid.Column** und **Grid.Row** auf die entsprechenden 0-basierten Werte einstellen.

Die gebundenen Steuerelemente

Wir haben drei verschiedene Arten gebundener Steuerelemente: **TextBox**, **ComboBox** und **DatePicker**. Das erste **TextBox**-Element heißt **txtID** und soll die Eigenschaft **ID** der Klasse **kunde** aufnehmen. Woher die Klasse **kunde** und die enthaltenen Eigenschaften bezogen werden, stellen wir später in der Code behind-Datei ein.

Hier ist zunächst wichtig, dass **kunde.ID** als Inhalt des Attributs **Text** innerhalb eines **Binding**-Elements angegeben wird:

```
Text="{Binding kunde.ID,
Mode=TwoWay,
NotifyOnValidationError=true,
ValidatesOnExceptions=true}"
```

Mit **Mode=TwoWay** legen wir außerdem fest, dass Änderungen nicht nur von der Eigenschaft zum Steuerelement, sondern auch in die umgekehrte Richtung weitergegeben werden. Um die übrigen beiden Attribute **NotifyOnValidationError** und **ValidatesOnExceptions** kümmern wir uns später.

Beim ersten **TextBox**-Element **txtID** haben wir außerdem das Attribut **IsEnabled** auf **False** eingestellt und **BorderBrush** auf **Transparent**. Dadurch kann der Benutzer den Inhalt nicht bearbeiten, der transparente Rahmen weist optisch darauf hin, dass es sich nicht um ein editierbares Feld handelt.

Die übrigen Textfelder referenzieren im Wert für das Attribut **Text** jeweils die entsprechende Eigenschaft der Klasse **kunde**, also etwa **kunde.Vorname**, **kunde.Nachname** und so weiter. Die Benennung der Textfelder beginnt jeweils mit **txt**.

ComboBox zur Auswahl der Anreden

Das **ComboBox**-Steuerelement **cboAnreden** soll die aktuelle Anrede des Kunden-Datensatzes anzeigen sowie die Auswahl der verfügbaren Anreden ermöglichen. Dazu binden wir es zunächst über das Attribut **ItemsSource** an eine Klasse namens **Anreden**. Der ausgewählte Eintrag wird über die Eigenschaft **SelectedItem** angegeben, welche die Eigenschaft **Anreden** des Objekts **kunde** referenziert. Das Attribut **DisplayMemberPath** gibt an, welche Eigenschaft im Kombinationsfeld angezeigt werden soll (**Bezeichnung**), **Selected-**

PropertyChanged in der Praxis

Der Artikel »Basics: PropertyChanged« zeigt, wie die Schnittstelle INotifyPropertyChanged funktioniert. In unserer Beispielanwendung Bestellverwaltung verwenden wir diese Schnittstelle an einer Stelle, wo wir einen Ribbon-Button in Abhängigkeit vom Wert einer Eigenschaft aktivieren und deaktivieren, die das PropertyChanged-Ereignis auslöst. Im vorliegenden Artikel zeigen wir, wie dies im Detail funktioniert.

Hintergrund

Unter Access war es üblich, den Zustand von Steuerelementen direkt per Code durch Ändern der entsprechenden Eigenschaft zu ändern. Das ist natürlich auch unter WPF/C# möglich. Allerdings gehört es dort zum guten Ton, die Definition der Benutzeroberfläche mit XAML und die Anwendungslogik in C#-Klassen voneinander zu trennen, und zwar in der Form, dass der Zustand der Benutzeroberfläche entweder direkt in XAML definiert wird oder aber durch die Bindung an Elemente der Anwendungslogik ermittelt wird. Keinesfalls jedoch sollten die Elemente der Anwendungslogik auf jene der Benutzeroberfläche zugreifen. Nun wollen wir in unserem Beispiel eine Schaltfläche im Ribbon, mit der man einen in einer Liste markierten Eintrag löschen kann, nur aktivieren, wenn die Liste auch im unteren Bereich des Fensters angezeigt wird. Ist das nicht der Fall, weil beispielsweise gerade die Details eines Kundendatensatzes dort abgebildet werden, soll die Schaltfläche deaktiviert sein. Unter Access hätte man nun ein Ereignis, das beim Einblenden der Kundenliste ausgelöst wird, genutzt, um die Schaltfläche zum Löschen eines Listeneintrags zu aktivieren oder zu deaktivieren.

Hier wollen wir nun aber moderner vorgehen und Benutzeroberfläche und Anwendungslogik so weit wie möglich voneinander trennen. Dazu legen wir im Code behind-Modul eine öffentliche Eigenschaft an, an die wir das **IsEnabled**-Attribut der betroffenen Ribbon-Schaltfläche binden. Den Wert dieser Eigenschaft stellen wir an einer geeigneten Stelle ein, in diesem Fall dem Ereignis **Navigated** des **Frame**-Objekts namens **Workzone**, das unsere verschiedenen **Page**-Objekte wie die Kundendetails oder die Kundenliste einblendet.

Umsetzung

Für die Umsetzung deklarieren wir zunächst eine **Boolean**-Variable namens **kundeLoeschbar**:

```
bool kundeLoeschbar;
```

Für diese legen wir eine öffentliche Eigenschaft namens **KundeLoeschbar** an und statten diese mit dem üblichen Getter und Setter aus. Dem Setter fügen wir gleich noch den Aufruf der Methode **OnPropertyChanged** hinzu und übergeben dieser mit einem neuen Objekt des Typs **PropertyChangedEventArgs** den Namen der Eigenschaft:

```
public bool KundeLoeschbar {
    get { return kundeLoeschbar; }
    set {
        kundeLoeschbar = value;
        OnPropertyChanged(new PropertyChangedEventArgs("KundeLoeschbar"));
    }
}
```

Um die Schnittstelle **INotifyPropertyChanged** zu implementieren, fügen wir das entsprechende Element zur Definition der Klasse **MainWindow** hinzu:

```
public partial class MainWindow : RibbonWindow, INotifyPropertyChanged {
```

Das einzige zu implementierende Element dieser Schnittstelle sieht so aus:

```
public event PropertyChangedEventHandler PropertyChanged;
```

Fehlt noch die Methode **OnPropertyChanged**, die zwischen den Setter und das Auslösen des Ereignisses gesetzt wird:

```
protected virtual void OnPropertyChanged(PropertyChangedEventArgs e) {
    if (PropertyChanged != null) {
        PropertyChanged(this, e);
    }
}
```

In der Ribbon-Definition stellen wir nun das Attribut **IsEnabled** so ein, dass es an die Eigenschaft **KundeLoeschbar** gebunden wird. **KundeLoeschbar** ist ein Element der Code behind-Klasse des **Window**-Elements. Diese Bindung können wir auf verschiedene Arten herstellen – wir stellen zwei davon vor. Die erste geht davon aus, dass Sie dem **Window**-Element einen Namen gegeben haben, in diesem Fall **wndMain**:

```
<RibbonWindow x:Class="Bestellverwaltung.MainWindow" ... Title="MainWindow" Height="450" Width="525" x:Name="wndMain">
```

Dann können wir **wndMain** als **ElementName** angeben und die Eigenschaft **KundeLoeschbar** als **Path**:

```
<RibbonButton Name="btnKundeLoeschen" Label="Kunde löschen" ... IsEnabled="{Binding ElementName=wndMain,
Path=KundeLoeschbar}"></RibbonButton>
```

AUSGABE 01/2017

DATENBANK ENTWICKLER

MAGAZIN FÜR DIE DATENBANKENTWICKLUNG MIT VISUAL STUDIO FÜR DESKTOP, WEB UND CO.

TOP-THEMEN:

WPF-BASICS	Namespaces	SEITE 3
VISUAL STUDIO	Quellcodeverwaltung mit Visual Studio und Git	SEITE 9
SQL SERVER	SQLite als Backend	SEITE 21
LÖSUNGEN	Bestellverwaltung planen	SEITE 32
LÖSUNGEN	EDM: 1:n-Beziehungen mit DataGrid	SEITE 44

WPF-Basics: Namespaces

Bislang haben wir immer mit WPF-Projekten gearbeitet, ohne uns Gedanken um das Stammelement einer .xaml-Datei zu machen. Dieses Element, das in der Regel Window heißt, enthält zum Beispiel einige Namespace-Angaben und weitere Informationen. In diesem Artikel wollen wir erläutern, was es mit diesen Attributen auf sich hat und wie sich diese auf die Definition der vollständigen .xaml-Datei auswirken.

Window oder ...

Wie schon erwähnt, ist das Stammelement einer **.xaml**-Datei meist ein **Window**-Element – also eines, das ein Fenster beschreibt. Es gibt allerdings noch einige andere Elementtypen, zum Beispiel das in diesem Magazin auch schon verwendete **Page**-Element, mit dem man eine Seite beschreibt, die innerhalb anderer Elemente eingebunden werden kann.

Wenn Sie eine neue WPF-Anwendung anlegen, erhalten Sie direkt ein Fenster mit dem Titel **MainWindow**, dass einige für den Einsteiger kryptisch anmutende Elemente enthält (siehe Listing 1).

Das erste Attribut namens **x:Class** gibt an, welche Klasse die Code behind-Klasse der aktuellen **.xaml**-Datei ist. Der Teil vor dem Punkt gibt den Namespace an, der Teil hinter dem Punkt den Klassennamen.

Im entsprechenden Code-Modul finden Sie als oberstes Element das **Namespace**-Objekt namens **Experimente**, darunter die Klasse mit dem unter **x:Class** angegebenen Namen **MainWindow**. Der Doppelpunkt und die folgende Bezeichnung **Window** geben an, dass die Klasse **MainWindow** von der Klasse **Window** abgeleitet ist und somit alle Eigenschaften, Methoden und Ereignisse der Klasse **Window** besitzt. Die Methode **public MainWindow** ist die sogenannte Konstruktor-Methode, die automatisch beim Initialisieren der Klasse aufgerufen wird:

```
namespace Experimente {
    public partial class MainWindow : Window {
        public MainWindow() {
            InitializeComponent();
        }
    }
}
```

Beim Starten wird in der Datei **App.xaml** die unter **StartupUri** angegebene **.xaml**-Datei eingelesen, hier **MainWindow.xaml**, die dortige unter **x:Class** angegebene Code behind-Klasse gesucht und die Konstruktor-Methode aufgerufen. Diese wiederum ruft die Methode **InitializeComponent** auf, die grob zusammengefasst aus der in der **.xaml**-Datei enthaltenen Definition das entsprechende Objekt zusammenstellt, also beispielsweise ein **Window**-Element mit all seinen Steuerelementen.

```
<Window x:Class="Experimente.MainWindow"
        xmlns="http://schemas.microsoft.com/winfx/2006/xaml/presentation"
        xmlns:x="http://schemas.microsoft.com/winfx/2006/xaml"
        xmlns:d="http://schemas.microsoft.com/expression/blend/2008"
        xmlns:mc="http://schemas.openxmlformats.org/markup-compatibility/2006"
        xmlns:local="clr-namespace:Experimente"
        mc:Ignorable="d"
        Title="MainWindow" Height="350" Width="525">
    <Grid>

    </Grid>
</Window>
```

Listing 1: Namespaces eines **Window**-Elements

xmlns-Attribute

Wozu sind nun die ganzen **xmlns…**-Attribute gut? Diese legen fest, welche Elemente Sie in der **.xaml**-Datei zur Definition des Aussehens etwa eines **Window**-Objekts nutzen können. Das erste **xmlns**-Attribut enthält den folgenden Wert, der jedoch nicht etwa tatsächlich eine Seite anzeigt, wenn Sie die URL in den Webbrowser eingeben:

```
xmlns="http://schemas.microsoft.com/winfx/2006/xaml/presentation"
```

Vermutlich kennen Sie schon die **using**-Anweisung, mit der Sie einer C#-Klasse die dort zu verwendenden Namespaces zuweisen und damit den Zugriff auf die dort enthaltenen Klassen ermöglichen.

Das oben genannte Attribut weist der **.xaml**-Datei nicht direkt einen Namespace zu wie es in einer C#-Klasse der Fall ist. Der Namespace **http://schemas.microsoft.com/winfx/2006/xaml/presentation** fasst aber einige der Klassen, die auch standardmäßig in einer C#-Klasse verfügbar gemacht werden, zusammen – zum Beispiel **System. Windows. System.Data** und noch einige mehr.

Wenn Sie dieses **xmlns**-Attribut einmal entfernen, werden Sie feststellen, dass beispielsweise das **Window**-Element in der **.xaml**-Datei als fehlerhaft unterstrichen wird. Der Fehler lautet in diesem Fall **The default namespace is not defined** (siehe Bild 1).

Wie in der Fehlermeldung ersichtlich, wird der hier fehlende Namespace auch als Standardnamespace bezeichnet. Wir benötigen diesen also, um Elemente wie **Window**, **Grid** und viele mehr für die Definition der Benutzeroberfläche heranzuziehen.

Der zweite Namespace heißt ähnlich wie der erste, aber er unterscheidet sich erstens in der Bezeichnung des Attributs, das noch ein durch einen Doppelpunkt angeführtes **x** enthält, und zweitens durch das Fehlen des letzten Teils **/presentation**:

```
xmlns:x="http://schemas.microsoft.com/winfx/2006/xaml"
```

Der Teil hinter dem Doppelpunkt, also das **x**, heißt in diesem Zusammenhang Präfix. Wenn Sie nun ein Element einer von diesem zweiten Namespace referenzierten Klassen im **.xaml**-Code verwenden wollen, das nicht im Standardnamespace enthalten ist, müssen Sie diesem ebenfalls das Präfix voranstellen. Das ist sogar schon im obigen **.xaml**-Code der Fall - das **Class**-Attribut des **Window**-Elements ist nämlich mit einem führenden **x** versehen:

Bild 1: Fehler durch den fehlenden Standardnamespace

IntelliSense-Unterstützung für eigene Code-Elemente

Wenn Sie die eingebauten Klassen, Eigenschaften, Methoden und Ereignisse nutzen, profitieren Sie oft von den per IntelliSense angezeigten Informationen über das jeweilige Element. Wenn Sie jedoch eigene Klassen mit ihren Membern programmieren, ist die IntelliSense-Unterstützung auf das Nötigste beschränkt. Aber wie soll es auch anders sein? Immerhin weiß Visual Studio ja nichts über den Zweck Ihrer Klassen, Methoden, Eigenschaften und Ereignisse. Also schauen wir uns an, wie Sie Ihren eigenen Code um IntelliSense-Texte erweitern.

IntelliSense-Texte hinzufügen

Die per IntelliSense angezeigten Texte lassen sich leicht hinzufügen, indem Sie entsprechende Auszeichnungen zum C#-Code hinzufügen.

Wenn Sie eine neue Klasse erstellen, finden Sie bereits eine solche Auszeichnung vor, die Sie bisher vielleicht eher als ablenkend betrachtet und direkt wieder gelöscht haben (siehe Bild 1).

Der zwischen dem öffnenden und schließenden **<summary>**-Element angegebene Text wird jedoch per IntelliSense eingeblendet, wenn Sie den Klassennamen eingeben und in der Liste der verfügbaren Elemente auswählen.

Wo können wir diese Auszeichnungen nun überall einsetzen? Den ersten Einsatzzweck haben Sie ja schon kennen gelernt – als Beschreibung einer Klasse. Um dies für eine neue Klasse zu erreichen, geben Sie in der Zeile direkt über der Klasse einfach drei Slash-Zeichen ein (**///**). Visual Studio erweitert dies automatisch wie folgt:

```
/// <summary>
///
/// </summary>
public class BeispielIntelliSense {

}
```

Sie müssen nun nur noch den anzuzeigenden Text zwischen die **<summary>**-Zeilen schreiben.

Mehrzeilige Texte

Wenn Sie nicht nur eine, sondern mehrere Zeilen mit Informationen hinzufügen möchten, liegt es nahe, einfach eine weitere Zeile hinzuzufügen – wieder mit führenden Slash-Zeichen.

Betätigen Sie am Ende der ersten Zeile die Eingabetaste, wird die Zeichenkette **///** sogar automatisch hinzugefügt.

Eine Erweiterung um eine zweite Zeile führt allerdings nicht automatisch zur Anzeige einer zweiten Zeile im IntelliSense-Text: Der gewünschte Zeilenumbruch wird dort nämlich unterschlagen.

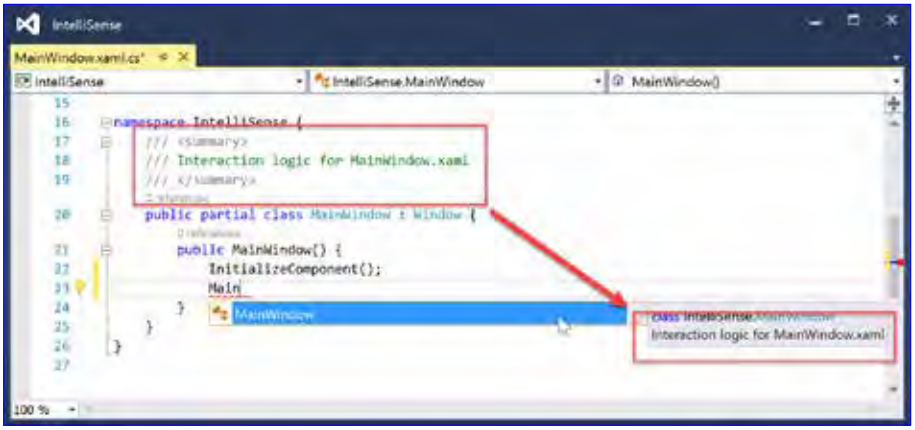


Bild 1: Automatisch hinzugefügte Code-Auszeichnung

Um mehrere Zeilen Text per IntelliSense anzuzeigen, müssen Sie jede Zeile in **<para>…</para>**-Elemente einfassen (siehe Bild 2).

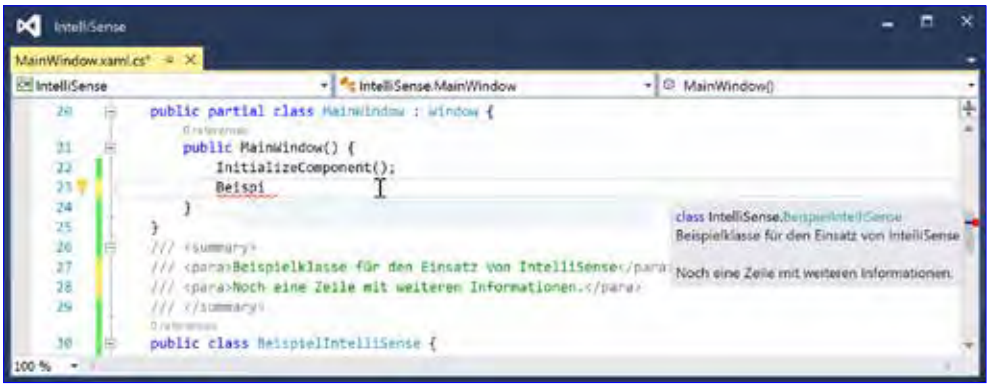


Bild 2: Beispiel mit zwei Zeilen

IntelliSense für Methoden

Auf die gleiche Weise fügen Sie die per IntelliSense anzuzeigenden Texte für Methoden oder andere Elemente hinzu. Bei Methoden ergeben sich noch einige zusätzliche Möglichkeiten, die Visual Studio automatisch unterstützt.

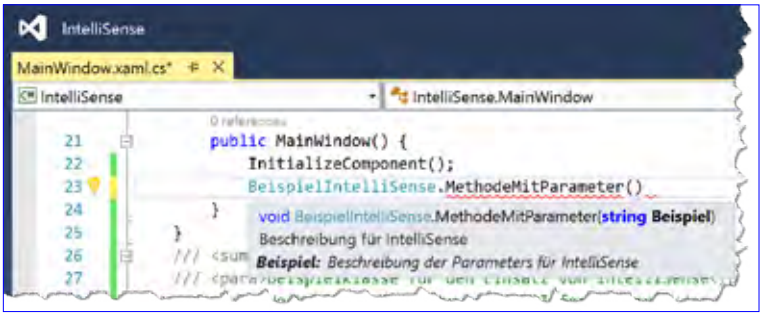


Bild 3: IntelliSense für die Parameter einer Methode

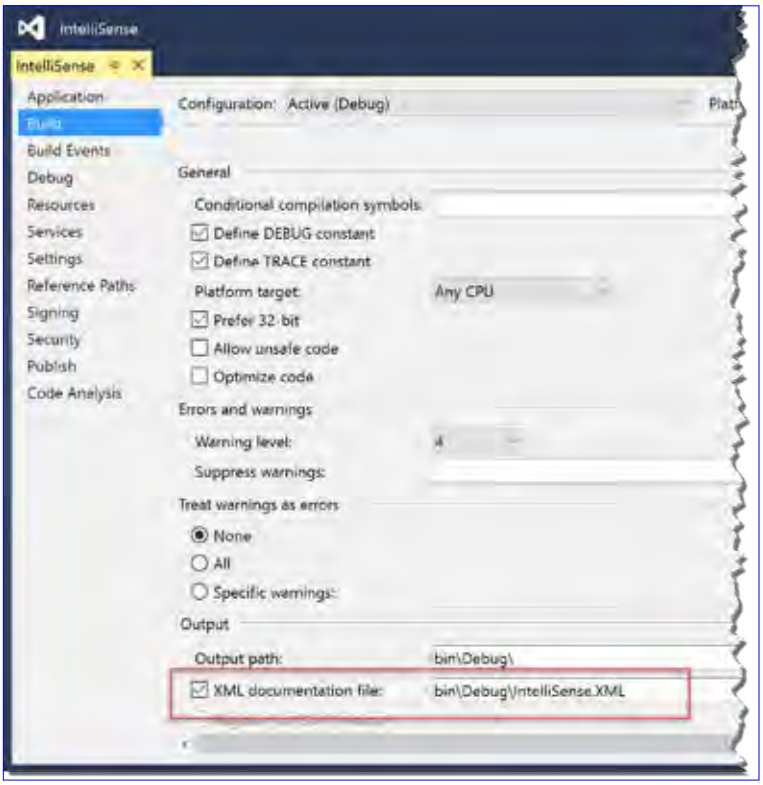


Bild 4: Festlegen des Speicherorts für die XML-Dokumentation

Im einfachen Fall einer Methode ohne Parameter sieht der Kommentar wie folgt aus:

```
/// <summary>
/// IntelliSense-Text für eine einfache Methode.
/// </summary>
public static void EinfacheMethode() {
    ...
```

Wenn Sie eine Methode mit einem oder mehreren Parametern anlegen und dann mit **///** die Auszeichnungen hinzufügen, erhalten Sie noch eine weitere Zeile, die wir in diesem Fall wie folgt füllen:

```
/// <summary>
/// Beschreibung für IntelliSense
/// </summary>
/// <param name="Beispiel">Beschreibung der
Parameters für IntelliSense</param>
public static void MethodeMitParameter(
    string Beispiel) {
    ...
```

Sie können also für die einzelnen Parameter auch noch Beschreibungen hinterlegen. Das ist

Quellcodeverwaltung mit Visual Studio und Git

Die Arbeit mit Visual Studio-Projekten bietet gegenüber Access einen interessanten Vorteil: Im Gegensatz zu Access, wo die einzelnen Objekte alle in der Access-Datei gespeichert wurden, finden Sie alle Objekte eines Visual Studio-Projekts als einzelne Dateien im Projektordner wieder. Das erleichtert vor allem die Verwaltung des Quellcodes in einer Quellcodeverwaltung. Eine solche erfasst nach Wunsch Zwischenstände der Entwicklung und bietet die Möglichkeit, später noch auf vorherige Stände zuzugreifen und diese wiederherzustellen, sollte sich die Programmierung in die falsche Richtung entwickelt haben. Dieser Artikel zeigt die Basics der Quellcodeverwaltung unter Visual Studio – vorerst für den Einsatz im stillen Kämmerlein eines Einzelentwicklers.

Bei einem üblichen WPF/C#-Projekt, wie Sie es im Rahmen dieses Magazins schon einige Male angelegt haben, taucht die Quellcodeverwaltung von Visual Studio nicht sichtbar in Erscheinung. Dies ist erst der Fall, wenn Sie das Kontextmenü der Projektmappe im Projektmappen-Explorer anzeigen – dort finden Sie den Eintrag **Projektmappe zur Quellcodeverwaltung hinzufügen** vor (siehe Bild 1).

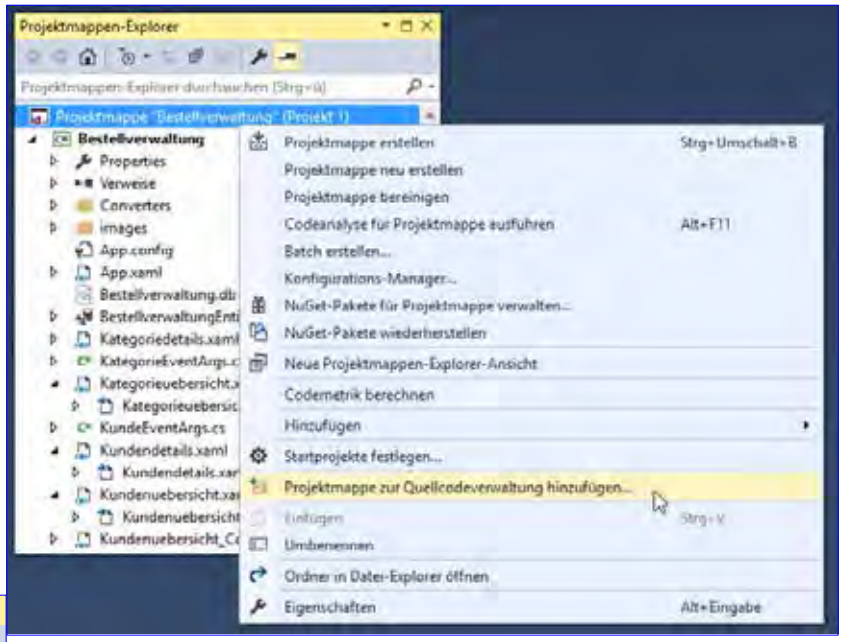


Bild 1: Hinzufügen der Projektmappe zu einer Quellcodeverwaltung

Nach dem Betätigen dieses Befehls geschieht nicht viel – außer dass die Dateien im Projektmappen-Explorer nun alle mit einem zusätzlichen Icon ausgestattet sind, dass beim Überfahren mit der Maus den Text **Eingecheckt** liefert (siehe Bild 2).

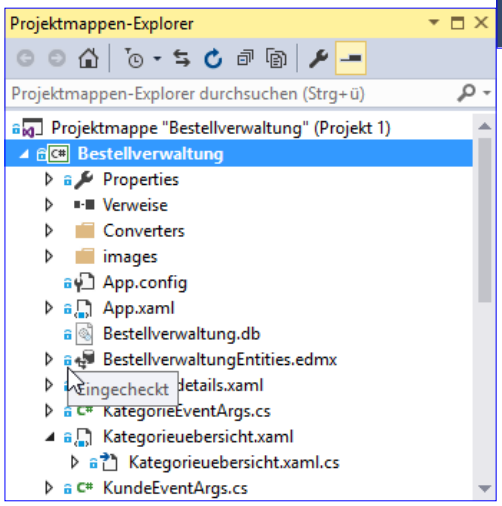


Bild 2: Nach dem Hinzufügen zur Quellcodeverwaltung

Das Hinzufügen der Projektmappe zur Quellcodeverwaltung sorgt also auch dafür, dass alle enthaltenen Dateien erstmalig eingecheckt werden. Einchecken bedeutet hierbei, dass die aktuelle Version der Dateien gespeichert wird. Dies war der Weg, um ein bestehendes Projekt zu einer Quellcodeverwaltung hinzuzufügen. Im Folgenden sehen wir uns allerdings die Vorgehensweise für ein neues, leeres WPF-Projekt an – dort werden wir dann ein paar Änderungen am Code vornehmen, einchecken, Änderungen betrachten, neue Dateien anlegen und mehr.

Neues Projekt direkt zur Quellcodeverwaltung hinzufügen

Wenn Sie über den Menüeintrag **Datei|Neu|Projekt...** ein neues Projekt erstellen, finden Sie im Dialog **Neues Projekt** rechts unten eine Option namens **Zur Quellcodeverwaltung hinzufügen** (siehe Bild 3). Aktivieren Sie diese Option und klicken dann auf **OK**, werden die neu erzeugten Projektdateien direkt zur Quellcodeverwaltung hinzugefügt.

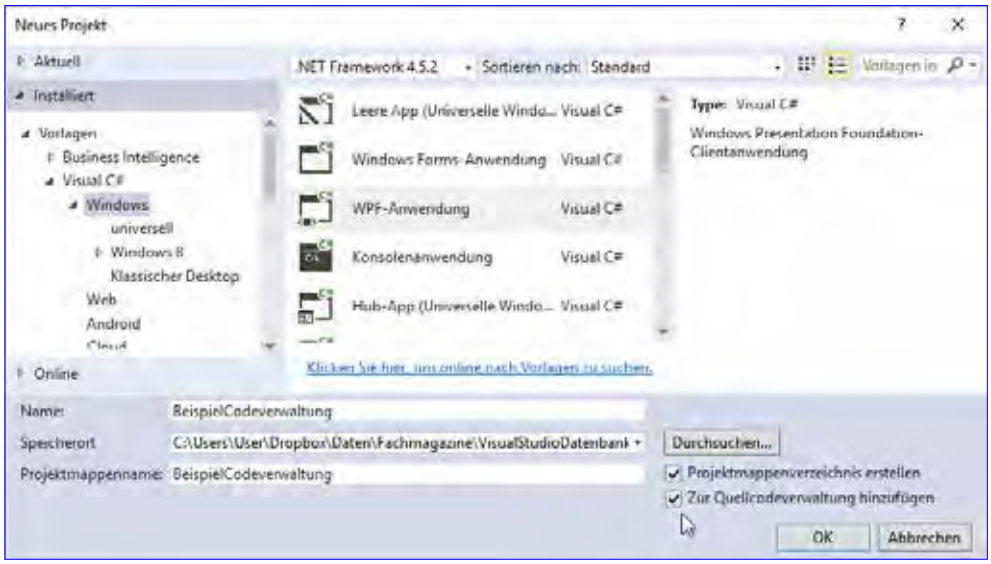


Bild 3: Anlegen eines neuen Projekts, das direkt zur Quellcodeverwaltung hinzugefügt werden soll

Der Projektmappen-Explorer zeigt dann direkt alle Dateien als eingecheckt an (siehe Bild 4).

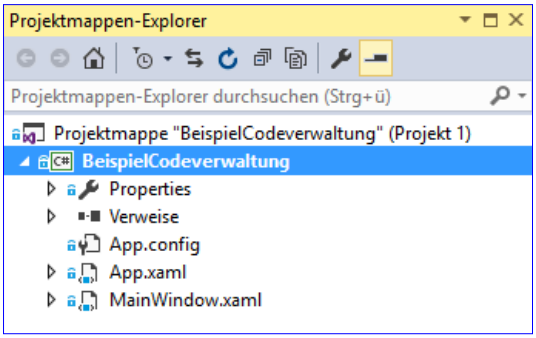


Bild 4: Ein frisches Objekt, direkt in die Quellcodeverwaltung eingecheckt

Bearbeiten einer Datei

Wenn Sie eine der eingecheckten und mit dem Schloss-Symbol versehenen Dateien bearbeiten, ändert sich das Symbol in einen roten Haken mit dem ToolTip-Text **Ausstehende Bearbeitung**.

Als Beispiel haben wir der noch unberührten Datei **MainWindow.xaml** ein neues Textfeld hinzugefügt, also von

```
<Grid>
</Grid>
```

zu

```
<Grid>
    <TextBox x:Name="txtBeispiel" Text="Beispiel"></TextBox>
</Grid>
```

Das Kontextmenü dieser Datei im Projektmappen-Explorer liefert nun einige weitere Optionen im Kontextmenü (siehe Bild 5):

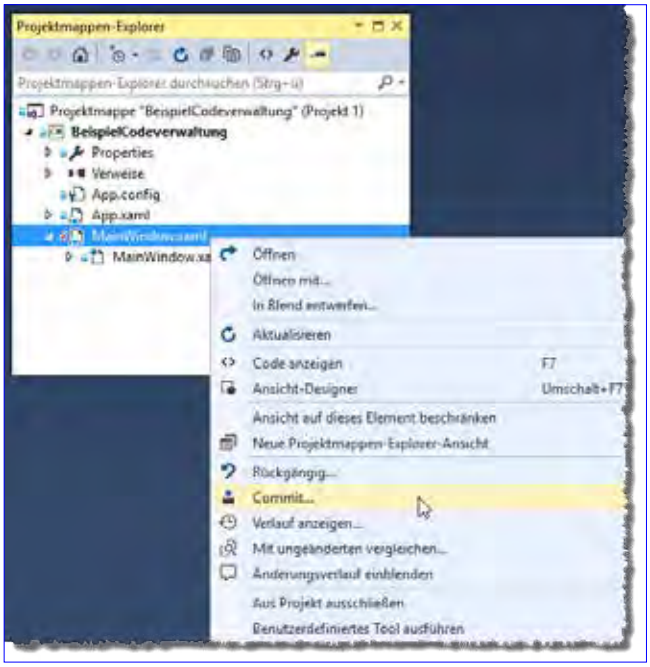


Bild 5: Optionen einer geänderten Datei

- **Commit**: Ruft den Bereich **Änderungen** auf, der alle seit dem letzten Commit durchgeführten Änderungen auflistet und die Möglichkeit bietet, eine Commitnachricht einzugeben. Dabei handelt es sich um einen benutzerdefinierten Text, mit dem Sie beispielsweise Informationen zu den getätigten Änderungen angeben können.

- **Verlauf anzeigen**: Zeigt eine Liste der bisherigen Commits für dieses Projekt an.

- **Mit ungeänderten vergleichen**: Öffnet einen Bereich, in dem die aktuelle, geänderte Version mit der Version vergleichen wird, wie sie zuvor aussah.

- **Änderungsverlauf einblenden**: Zeigt alle Änderungen in einer Ansicht an.

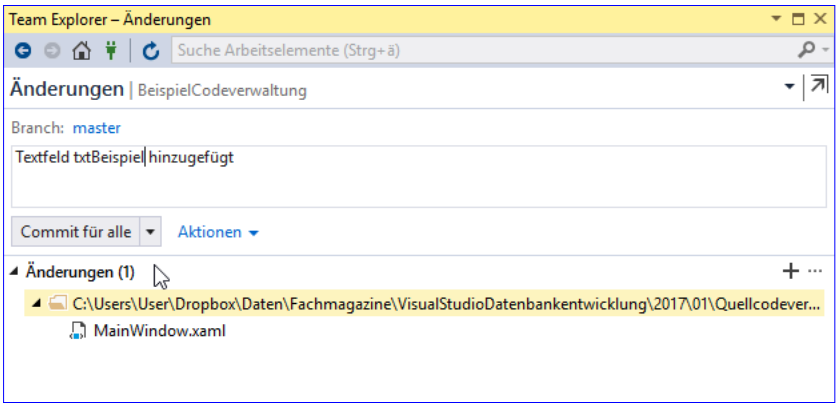


Bild 6: Angaben vor dem Commit

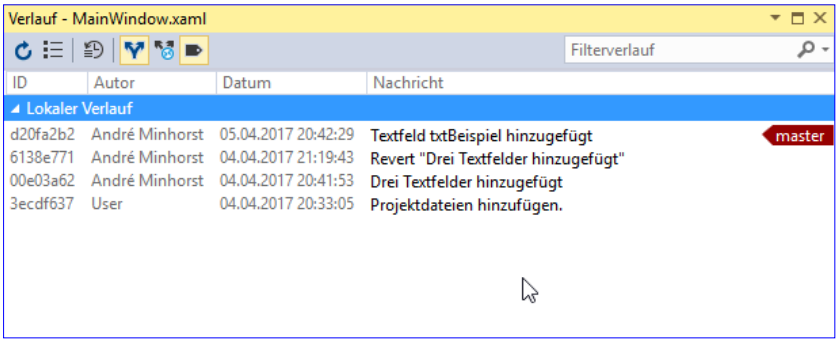


Bild 7: Änderungsverlauf einer Datei

Diese Bereiche schauen wir uns in den folgenden Abschnitten im Detail an.

Änderungen-Dialog

Der Bereich **Team-Explorer - Änderungen** zeigt jeweils die seit dem letzten Commit geänderten Dateien in einer Liste im unteren Bereich an. Diesen Bereich können Sie immer mit dem Menübefehl **Ansicht|Team Explorer** oder mit der Tastenkombination **Strg + ^, Strg + M** einblenden. Sobald eine Änderung vorliegt, können Sie eine Commitnachricht eingeben und die Änderungen mit einem Klick auf die Schaltfläche **Commit für alle** durchführen (siehe Bild 6). Das erste offensichtliche Resultat ist, dass das Icon der geänderten Dateien wieder vom roten Haken in das blaue Schloss geändert wird. Die Schaltfläche **Commit für alle** wird übrigens nur bei Vorliegen von Änderungen und bei Vorhandensein einer Commitnachricht aktiviert.

Die Schaltfläche bietet übrigens noch weitere Funktionen an, wenn Sie auf den Pfeil nach unten klicken:

- **Commit für alle**: Überträgt die Änderungen in das lokale Repository.

- **Commit für alle und Push**: Überträgt die Änderungen in das lokale Repository und »pusht« diese auch gleich an eventuell angeschlossene Remote-Repositorys (weitere Informationen weiter unten).

- **Commit für alle und Sync**: Überträgt die Änderungen in das lokale Repository, »synct« die Änderungen von einem eventuell angeschlossenen Remote-Repository und »pusht« die Änderungen dann in das Remote-Repository. Sollten dabei an der

Schnellzugriff über die Statusleiste

Die Statusleiste bietet unten rechts einige Möglichkeiten für den Schnellzugriff auf die Elemente der Quellcodeverwaltung (siehe Bild 16). Mit dem Menü ganz rechts erhalten Sie Befehle zum Anlegen eines neuen Branches, zum Öffnen der Branch-Übersicht, zum Anzeigen des Verlaufs und zum schnellen Auschecken der verfügbaren Branches. Mit dem Befehl links daneben öffnen Sie den Team Explorer. Die Zahl neben dem Stift-Symbol liefert die Anzahl der ausstehenden und noch nicht eingecheckten Änderungen. Klicken Sie auf dieses Symbol, erscheint der Team Explorer mit der **Änderungen**-Ansicht, mit der Sie einen Commit ausführen können. Ganz links folgt noch die Anzahl der unveröffentlichten Commits. Unveröffentlicht bedeutet, dass die Commits noch nicht an einen Git-Service übermittelt wurden. Was das ist, erläutern wir im folgenden Abschnitt.

Git-Service nutzen

Bisher haben wir die Commits nur im lokalen Git-Repository gespeichert. Das heißt, dass Sie die Änderungen am lokalen Rechner gespeichert haben und darauf zugreifen können. Wenn Sie nun mit mehreren Entwicklern am gleichen Projekt arbeiten möchten, bietet es sich an, einen Git-Service zu nutzen. Damit erstellen Sie ein Repository auf einem Git-Server, der zum Beispiel von den Visual Studio Team Services bereitgestellt wird.

Diesen Dienst richten Sie ein, indem Sie den Team Explorer aktivieren und dort zur Seite **Sync** wechseln (siehe Bild 17). Hier haben Sie zwei Möglichkeiten: die Nutzung der Team Services von Microsoft oder die Angabe eines alternativen Repositories.

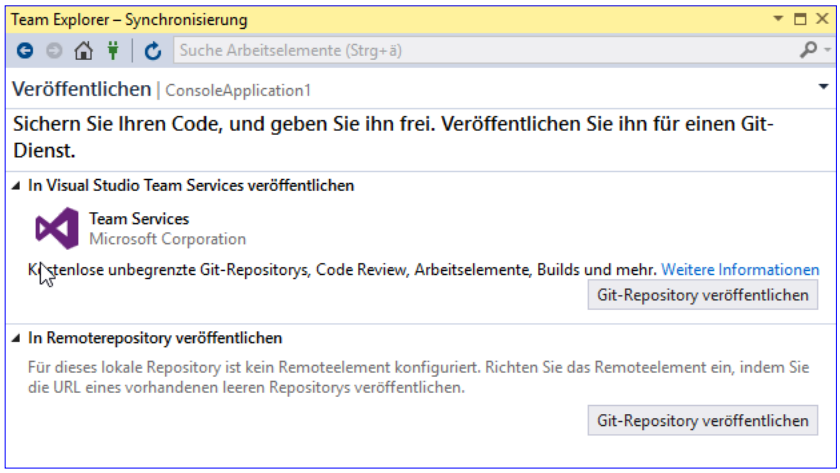


Bild 17: Einrichten eines Git-Repositories mit einem Git-Service

Wir wollen der Einfachheit halber die erste Variante vorstellen. Dazu klicken Sie auf die obere der beiden Schaltflächen mit der Beschriftung **Git-Repository veröffentlichen**. Im folgenden Schritt wählen Sie Ihr Microsoft-Konto aus dem nun erscheinenden Kombinationsfeld aus oder fügen ein Konto hinzu (siehe Bild 18).

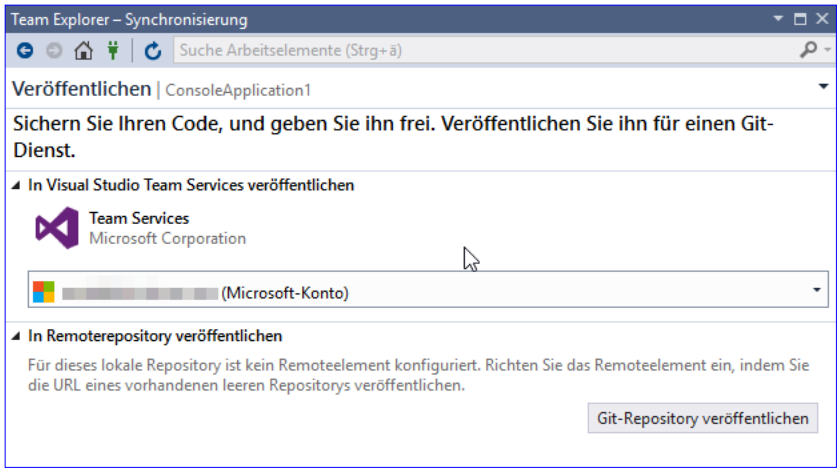


Bild 18: Angabe des Microsoft-Kontos

Aber warum geschieht nun nichts? Ganz einfach: Weil Sie vermutlich noch nicht die Visual Studio Team Services für Ihr Microsoft-Konto freigeschaltet haben. In diesem Fall suchen Sie die folgende Seite auf und starten die Visual Studio Team Services für Ihr Microsoft-Konto:

```
https://www.visualstudio.com/de/team-services
```

Da sich diese Seiten regelmäßig ändern, hier nur die prinzipielle Vorgehensweise für die Einrichtung: Als Erstes geben Sie an, unter welcher URL Ihre Projekte gehostet werden sollen. Die URL folgt dem Schema **<Name>.visualstudio.com**, wobei Sie den Teil für den Platzhalter **<Name>** selbst festlegen können (da der Name eindeutig sein muss, kann es vorkommen, dass Sie hier mehrere Versuche benötigen). Dann legen Sie fest, ob der Code mit **Git** oder der **Team Foundation-Versionskontrolle** gehostet werden soll. Hier wählen wir **Git** aus. Außerdem geben Sie hier gegebenenfalls noch einen Projektnamen an, legen fest, wie die Arbeit organisiert werden soll und wo die Projekte gehostet werden sollen (**Westeuropa**). Danach wird das Konto erstellt, was ein paar Sekunden dauert.

Danach landen Sie in der Quellcodeverwaltung für Ihr neues Projekt, dass nun natürlich noch nicht mit einem lokalen Repository auf Ihrem Rechner verknüpft ist.

Wenn Sie nun zu Visual Studio zurückkehren, den Team Explorer schließen, diesen erneut öffnen und dann zum Bereich **Sync** wechseln, sollte unter der Auswahl Ihres Windows-Kontos die soeben von Ihnen angelegte Team-Services-Domäne erscheinen. Sie können hier nun auf die Schaltfläche **Erweitern** klicken und erhalten die Ansicht aus Bild 19.

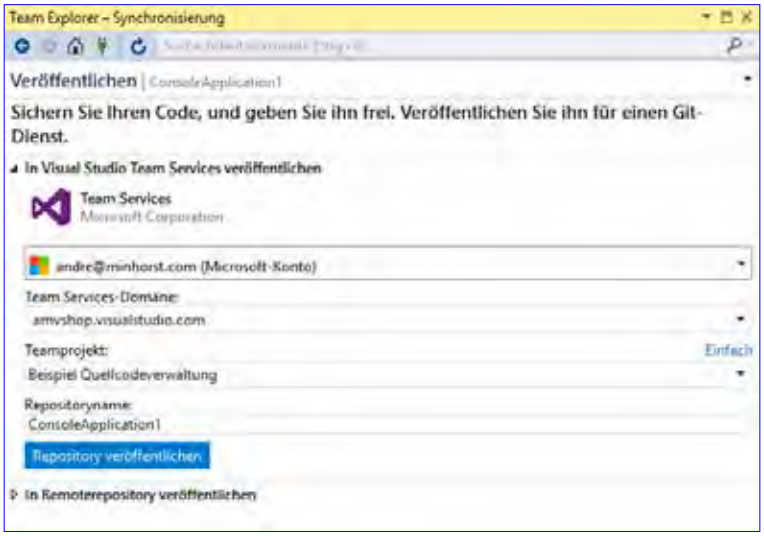


Bild 19: Veröffentlichen des Repositories

Hier finden Sie nun noch den Namen des soeben online festgelegten Teamprojekts und können einen Namen für das Repository angeben. Ein Klick auf die Schaltfläche **Repository** veröffentlichen startet die Veröffentlichung, die schließlich im Team Explorer als erfolgreich vermeldet wird. Mit dem Link **Im Web anzeigen** landen Sie schließlich auf Ihrer Team Services-Seite von Microsoft (siehe Bild 20).

Hier finden Sie nun allerdings nur den einen Branch vor, den Sie soeben veröffentlicht haben. Wenn Sie auch die anderen Branches veröffentlichen wollen, müssen Sie diese im Team Explorer anzeigen und diese mit dem Kontextmenü-Befehl **Branch veröffentlichen** zum Remote-Repository schicken.

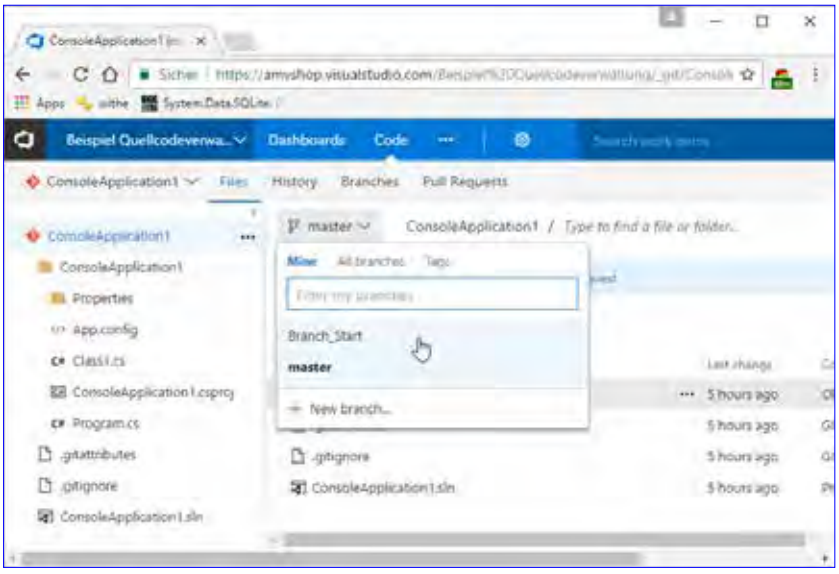


Bild 20: Das Repository mit den zwei Branches im Web-Browser

Sie können die im Remote-Repository befindlichen Dateien nun sogar vom Web-Browser aus einsehen und bearbeiten

SQLite als Backend

Der SQL Server kommt als mächtiges Installationspaket, und selbst die schlanke Variante LocalDb muss erst einmal installiert werden, bevor die Anwendung auf dem Zielsystem auf Daten zugreifen kann. Schlank und ohne eigene Installation – das wären gute Eigenschaften für ein Datenbanksystem für den Einsatz in einfachen Desktop-Anwendungen. Die Lösung lautet SQLite: Diese Programmbibliothek können Sie einfach zum Projekt hinzufügen und es mit der Anwendung auf dem Zielrechner installieren. Und das Beste: SQLite unterstützt sogar das Entity Framework.

SQLite – wozu?

Unser primärer Anlass, einmal nach Alternativen zum SQL Server oder zu LocalDb zu suchen, sind die Beispielprojekte zu diesem Magazin. Wenn diese eine Datenbank enthalten, sind immer der eine oder andere Schritt nötig, bevor Sie die Beispiele auf Ihrem Rechner ausprobieren können. Und das ist selbst der Fall, wenn bereits eines der Microsoft-Datenbanksysteme auf ihrem Rechner installiert ist. Welche Voraussetzungen sollte das gesuchte Datenbanksystem also haben? Am wichtigsten ist natürlich, dass wir damit weiterhin die Techniken vorstellen können, die Sie auch bei Verwendung von Microsoft-Datenbanken nutzen – also Beispielsweise der Einsatz des Entity Framework als Schnittstelle zwischen Anwendung und Datenbank. Der zweite Punkt ist, dass eine möglichst einfache Installation der Programmdateien des Datenbanksystems möglich ist oder am besten gar keine. Stattdessen sollte das Datenbanksystem am besten direkt im Projektordner an den Zielrechner übergeben werden und bereits fertig konfiguriert sein.

Wir hätten uns natürlich auch mit einer Access-Datenbank als Backend zufrieden gegeben, denn Access dürfte bei der Zielgruppe dieses Magazins auf dem Rechner installiert sein. Allerdings erfüllt eine Access-Datei gleich zwei Punkte nicht: Erstens ist für die Verwendung immer noch eine Access-Installation auf dem Zielrechner nötig. Dies ist zwar aufgrund der Runtime-Version, die kostenlos weitergegeben werden darf, möglich, aber besser ist natürlich ein Datenbanksystem, das komplett ohne Installation auskommt. Zweitens können wir zwar per ADO.NET auf Access-Datenbanken zugreifen, aber spätestens, wenn das Entity Framework zum Einsatz kommen soll (und das Entity Framework wird uns in diesem Magazin noch eine Weile begleiten), wird es eng – es gibt nämlich keine Implementierung, die den Zugriff auf die Tabellen einer Access-Datenbank erlaubt. Darüber sind wir dann auf SQLite gestoßen. Diese Datenbank kommt im Wesentlichen in Form einer Programmbibliothek, die außerdem auch noch nur wenige hundert Kilobyte groß ist. Und es hat eine weitere Gemeinsamkeit mit Access: Die eigentlichen Datenbanken kommen in Form einer einzigen Datei, in diesem Fall mit der Dateiendung **.db**.

SQLite ist nicht von ungefähr so kompakt: In der Regel wird es etwa in Apps für Mobiltelefone eingesetzt. Die Kompaktheit hat natürlich auch ihren Preis: SQLite enthält längst nicht alle Funktionen, die ein »großes« Datenbanksystem wie etwa SQL Server bietet. Für den Einsatz in unseren Beispieldatenbanken reichen die Funktionen aber allemal aus. In den folgenden Abschnitten erfahren Sie verschiedene Dinge über SQLite:

- mit welchem Tool Sie SQLite-Datenbanken erstellen und bearbeiten können,
- wie Sie SQL Server-Datenbanken nach SQLite migrieren,
- wie Sie SQLite in .NET-Projekten einsetzen und ein Entity Data Model auf Basis einer SQLite-Datenbank erstellen und
- wie Sie eine Anwendung mit SQLite-Datenbank weitergeben können.

SQLite-Tools

Bevor wir überhaupt mit unserer Anwendung auf eine SQLite-Datenbank zugreifen können, sollten wir uns eine solche erstellen. Dazu gibt es verschiedene Tools. Wir haben den **DB Browser for SQLite** verwendet, den Sie unter folgendem Link finden:

```
http://sqlitebrowser.org/
```

Hier laden Sie einfach den Download unter dem Link **Windows .exe** in der 32bit- oder 64bit-Version herunter und installieren die Anwendung.

Nach der Installation können Sie das Tool direkt starten. Hier bietet sich auch direkt die Schaltfläche **Neue Datenbank** zum Erstellen einer neuen Datenbank an. Wir wollen aber nicht tiefer in den Umgang mit diesem Tool einsteigen, sondern dieses nur kurz vorstellen – also Notlösung, wenn Sie einmal in den Entwurf einer Datenbank eingreifen müssen. Wir wollen aber dennoch kurz einmal einen Blick in eine echte SQLite-Datenbank werfen. Eine solche findet sich auch recht schnell, und zwar unter dem folgenden Link:

```
http://www.sqlitetutorial.net/sqlite-sample-database/
```

Nach dem Download finden Sie eine Datei namens **chinook.db** im Zielverzeichnis. Öffnen Sie diese nun mit dem **Tool DB Browser for SQLite**, füllt sich dieses recht schnell mit bekannt aussehenden Elementen wie Tabellen, Feldern und Felddatentypen (siehe Bild 1).

Vom SQL Server zu SQLite

Sie können diese Tabelle als Ausgangspunkt für eigene Experimente nutzen, aber wir wollen gern mit der bereits in weiteren Beispielen verwendeten Datenbank Bestellverwaltung weiter arbeiten. Wie aber machen wir aus einer SQL Server-Datenbank eine SQLite-Datenbank? Auch hierfür gibt es ein Tool, dass Sie unter diesem Link finden:

```
https://www.codeproject.com/articles/26932/convert-sql-server-db-to-sqlite-db
```

Hier müssen Sie sich allerdings anmelden, um dann die gewünschte Version, hier wohl die Binary, herunterzuladen. Sie erhalten hier eine **.zip**-Datei mit einem Verzeichnis namens **SqlConverter_v1_20**, dass Sie einfach auf Ihre Festplatte kopieren. Starten Sie die Anwendung, die als **Converter.exe** in diesem Verzeichnis steckt, erscheint das Fenster aus Bild 2.

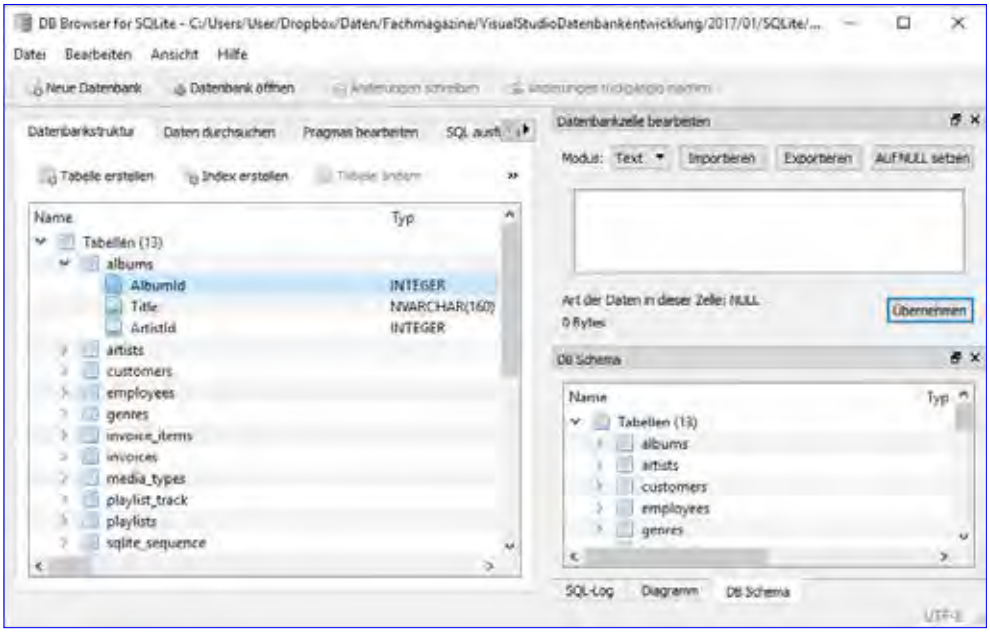


Bild 1: Fenster des SQLite-Tools **DB Browser for SQLite**

Geben Sie hier den Servernamen des SQL Servers ein und wählen Sie dann die Quelldatenbank aus. Legen Sie außerdem den Namen der zu erstellenden **.db**-Datei fest, zum Beispiel **Bestellverwaltung_SQLite.db**. Ein Mausklick auf die Schaltfläche **Start the conversion** startet den Vorgang, bei dem Sie zunächst mit dem Dialog aus Bild 3 die zu konvertierenden Tabellen festlegen.

Danach dürfte eine Datei namens **Bestellverwal-**

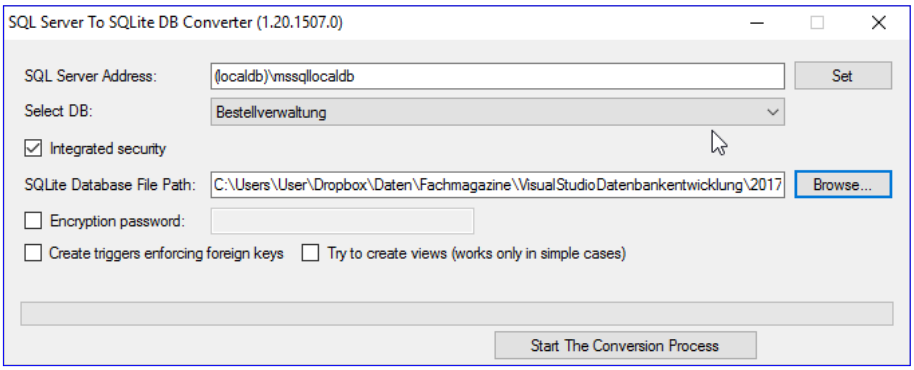


Bild 2: Tool zum Konvertieren von SQL Server-Datenbanken nach SQLite

tung_SQLite.db im gewählten Zielverzeichnis gelandet sein. Öffnen Sie diese Datei nun mit dem **DB Browser for SQLite**, können Sie schnell prüfen, ob alle Tabellen und Felder wie gewünscht in der Zieldatenbank gelandet sind (siehe Bild 4).

Vom Artikel zum Produkt

Wir wollen in diesem Rahmen auch gleich eine kleine Änderung an der Beispieldatenbank vornehmen: Im Gegensatz zu den englischen Tabellenbezeichnungen, die sich im Plural gegenüber dem Singular durch ein zusätzliches, angehängtes s unterscheiden (**customer** – **customers**), ist es im deutschen Sprachgebrauch nicht so einfach. Dummerweise schleichen sich hier mitunter Bezeichnungen wie **Artikel** ein, die im Singular wie im Plural gleich lauten. Um hier keine Probleme zu bekommen, wollen wir die Bezeichnung der SQL Server-Tabelle **Artikel** zuvor in **Produkte** ändern. Außerdem fallen entsprechende Änderungen in der Tabelle **Bestelldetails** an, das ja die Tabelle **Artikel**/**Produkte** per Fremdschlüsselfeld referenziert.

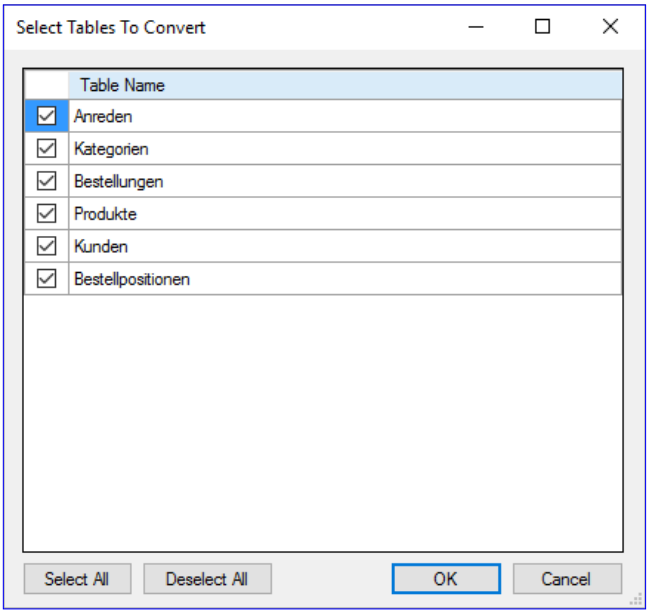


Bild 3: Auswahl der zu konvertierenden Tabellen

SQLite in .NET-Projekten

Wie greifen wir nun von einer .NET-Anwendung auf die Daten einer SQLite-Datenbank zu? Da wir den Fokus aktuell auf den Zugriff per Entity Framework setzen, ist dies unser Ansatz für den Zugriff auf die soeben umgewandelte Datenbank.

SQLite mit Entity Framework

Wir wollen also zunächst ein Entity Data Model auf Basis der SQLite-Datenbank erstellen. Dazu erstellen wir ein neues Projekt auf Basis der Vorlage **Visual C#|Windows|WPF**-

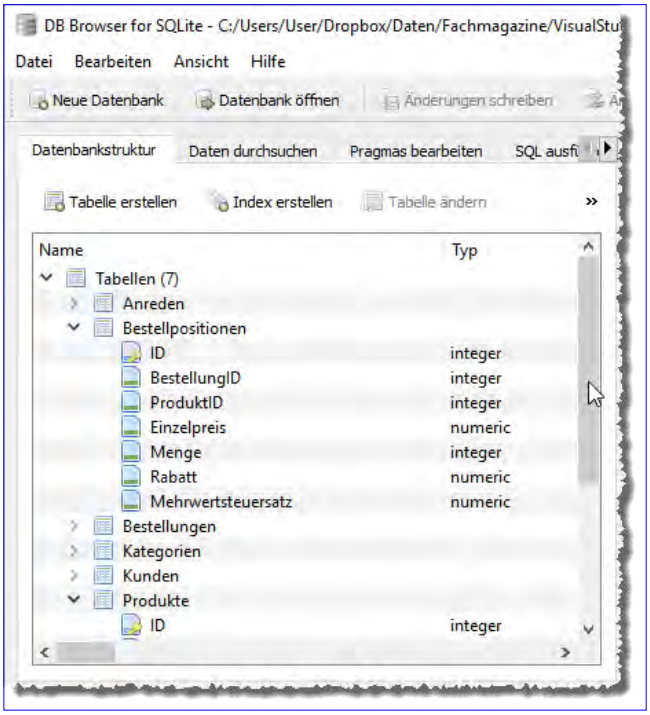


Bild 4: Anzeige der konvertierten Datei im **DB Browser for SQLite**

Anwendung namens **Bestellverwaltung_SQLite**.

Anschließend würden wir bei Verwendung des SQL Servers einfach ein neues Element des Typs **ADO.NET Entity Data Model** hinzufügen. Das geht nun nicht direkt, sondern wir müssen einen kleinen Umweg gehen. Wir müssen nämlich zuerst ein entsprechendes NuGet-Paket hinzufügen.

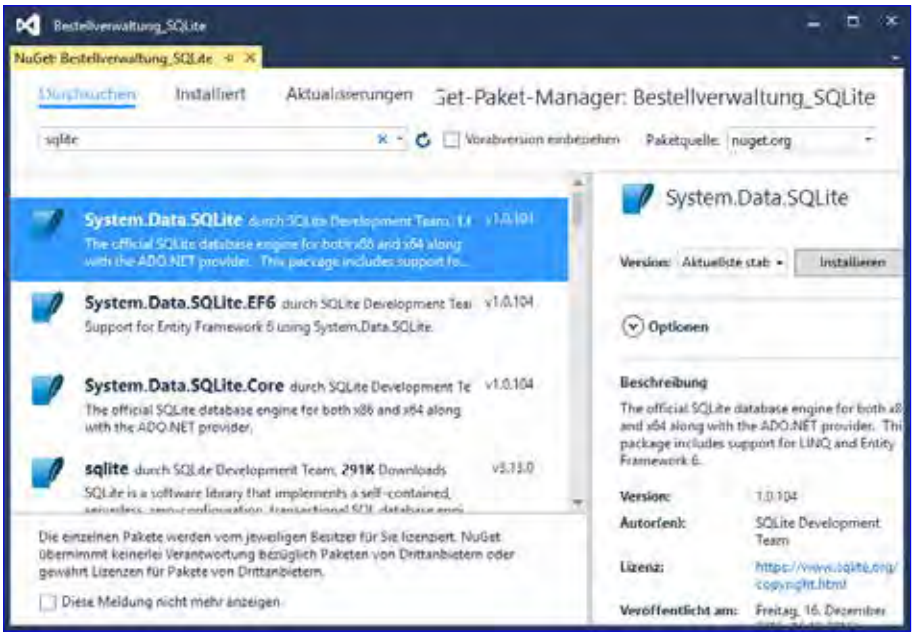


Bild 5: Hinzufügen des NuGet-Pakets für SQLite

SQLite-Erweiterung zum Projekt hinzufügen

Dazu wählen Sie den Menüeintrag **Projekt|Nuget-Pakete verwalten...** aus und wechseln im nun erscheinenden Dialog auf die Seite **Durchsuchen**.

Hier geben Sie als Suchbegriff **SQLite** ein und finden dann recht schnell das Paket **System.Data.SQLite**, welches alles enthält, was wir benötigen (siehe Bild 5). Wählen Sie diesen Eintrag aus und klicken Sie auf die Schaltfläche **Installieren**.

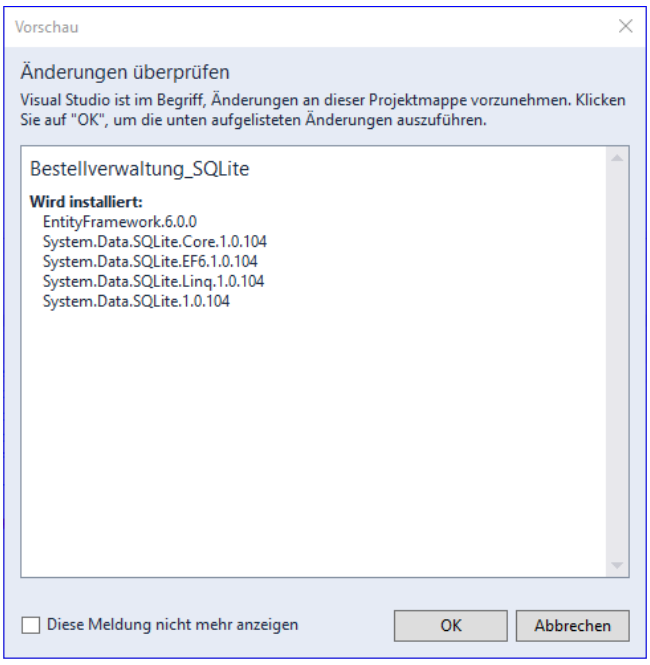


Bild 6: Bestätigen der Änderungen

Nach wenigen Augenblicken und dem Bestätigen der Meldung aus Bild 6 finden Sie im Projektmappen-Explorer einige neue Einträge unter Verweise, nämlich **System.Data.SQLite**, **System.Data.SQLite.EF6** und **System.Data.SQLite.Linq**. Damit haben Sie sich nicht nur Entity Framework-, sondern gleich auch noch Linq-Unterstützung ins Projekt geholt.

Entity Data Model hinzufügen

Damit können wir nun ein Entity Data Model auf Basis der SQLite-Datenbank erstellen. Öffnen Sie den Dialog **Neues Element hinzufügen** (zum Beispiel mit der Tastenkombination **Strg + Umschalt + A**) und wählen Sie dort den Eintrag **ADO.NET Entity Data Model** aus. Im Dialog **Neues Element hinzufügen** geben Sie als Namen für das Entity Data Model den Wert **BestellverwaltungEntities** an. Klicken Sie dann auf Hinzufügen, erscheint der **Assistent für Entity Data Model**. Hier wählen Sie den Eintrag **EF Designer aus Datenbank** aus.

Nun folgt der Schritt **Wählen Sie Ihr Datenverbindung aus**. Hier klicken Sie auf **Neue Verbindung** und öffnen damit den

Anwendung von SQL Server zu SQLite wechseln

Die aktuelle Beispieldatenbank zu diesem Magazin namens Bestellverwaltung verwendet bisher eine SQL Server-Datenbank gleichen Namens als Datenbank. Nun wollen wir für kommende Beispiele auf SQLite als Datenbanksystem wechsen, da es schlanker und für Beispiele perfekt geeignet ist. In diesem Zuge macht es Sinn, einmal zu betrachten, wie Sie das Entity Data Model für Ihre Datenbankanwendung wechseln und die Anwendung anschließend wie zuvor weiter verwenden können.

Die Ausgangsposition ist also die Beispieldatenbank, die wir in Ausgabe 6/2016 zuletzt aktualisiert haben und die per Ribbon die Bearbeitung von Kunden ermöglicht. Wie können wir für diese Datenbankanwendung, die auf einem Entity Data Model auf Bais der SQL Server-Datenbank **Bestellverwaltung** basiert, ein Entity Data Model auf Basis einer anderen Datenbank eines anderen Datenbanksystems erstellen?

Als Erstes fügen wir, weil wir mit SQLite arbeiten wollen, mit dem NuGet-Paket-Manager das Paket **System.Data.SQLite** zum Projekt hinzu (Vorgehensweise hierzu und Details zu den folgenden Schritten siehe Artikel **SQLite als Backend**).

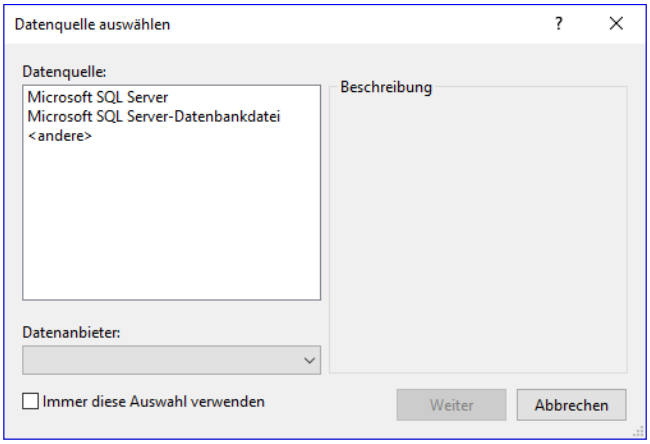


Bild 1: Der SQLite-Provider fehlt.

Prüfen, ob SQLite-Provider da ist

Die nächsten Schritte probieren Sie zunächst einmal aus, um zu ermitteln, ob der SQLite-Provider verfügbar ist – falls ja, entfernen wir vor dem tatsächlichen Hinzufügen des Entity Data Models auf Basis der SQLite-Datenbank anschließend erst das vorhandene Entity Data Model. Zum Testen fügen wir über den Dialog **Neues Element hinzufügen** (Menüeintrag **ProjektlNeues Element hinzufügen**) ein neues **Entity Data Model** namens **BestellverwaltungEntities** hinzu. Beim Schritt **Wählen Sie Ihre Datenverbindung aus** klicken Sie auf **Neue Verbindung...** und finden dann den Dialog aus Bild 1 vor, der den erwarteten SQLite-Eintrag nun eventuell nicht anzeigt.

In diesem Fall gehen Sie wie folgt vor:

- Öffnen Sie die Datei **App.config**.
- Suchen Sie den folgenden Eintrag und entfernen Sie den kursiv gedruckten Teil:

```
<?xml version="1.0" encoding="utf-8"?>
<configuration>
  ...
  <system.data>
    <DbProviderFactories>
```

```
      <remove invariant="System.Data.SQLite.EF6" />
      <add name="SQLite Data Provider (Entity Framework 6)" ...
          type="System.Data.SQLite.EF6.SQLiteProviderFactory, System.Data.SQLite.EF6" />
      <remove invariant="System.Data.SQLite" />
      <add name="SQLite Data Provider" ... type="System.Data.SQLite.SQLiteFactory, System.Data.SQLite" />
    </DbProviderFactories>
  </system.data>
</configuration>
```

- Kompilieren Sie die Anwendung.
- Versuchen Sie erneut, das Entity Data Model hinzuzufügen, was diesmal gelingen sollte.

Gelingt es immer noch nicht, haben Sie möglicherweise noch nicht alle nötigen Komponenten installiert – siehe Artikel **SQLite als Backend**. Ob dies der Fall ist, können Sie auch testen, indem Sie den Menüeintrag **Extras|Mit Datenbank verbinden...** aufrufen und im Dialog **Verbindung hinzufügen auf Ändern...** klicken. Sollte dann im nächsten Dialog wie in Bild 2 der Eintrag **System.Data.SQLite Database File** auftauchen, ist der Provider jedenfalls vorhanden.

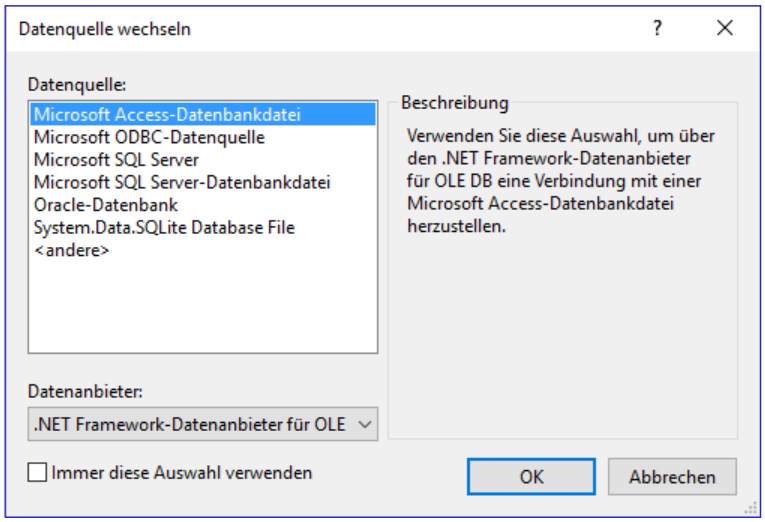


Bild 2: SQLite ist grundsätzlich verfügbar.

SQL Server-EDM löschen

Danach gehen wir recht rustikal vor und löschen den alten Eintrag **BestellverwaltungModel.edmx** einfach aus dem Projektmappen-Explorer, indem wir den Zweig **BestellverwaltungModel.edmx** entfernen. Danach löschen wir noch den folgenden Teil aus der Datei **App.config**:

```
<connectionStrings>
  <add name="BestellverwaltungEntities" connectionString="metadata=res://*/BestellverwaltungModel.csdl|res://*/Be-
stellverwaltungModel.ssdl|res://*/BestellverwaltungModel.msl;provider=System.Data.SqlClient;provider connection
string=&quot;data source=(localdb)\MSSQLLocalDB;initial catalog=Bestellverwaltung;integrated security=True;MultipleActive
ResultSets=True;App=EntityFramework&quot;" providerName="System.Data.EntityClient" />
</connectionStrings>
```

SQLite-EDM hinzufügen

Erst dann legen wir tatsächlich das Entity Data Model auf Basis der SQLite-Datenbank **Bestellverwaltung.db** an. Dieser Vorgang resultiert in der Anzeige des Diagramms der Datei **BestellverwaltungEntities.edmx**. Hier ändern Sie noch die Bezeichnungen der Entitäten, sodass diese im Singular statt im Plural angegeben sind.

Bestellverwaltung planen

Unsere Bestellverwaltung wächst weiter. Wir haben bereits eine Übersicht der Kunden und eine Detailansicht, mit der Sie Kunden anzeigen oder neue Kunden anlegen können. Nun sollen noch Übersichten und Detailansichten für Produkte, Kategorien und weitere Elemente hinzukommen. Hier sind nun einige grundsätzliche Entscheidungen bezüglich des Aufbaus der Anwendung zu treffen – zum Beispiel, wie das Ribbon erweitert werden soll oder wie nach dem Ändern oder Anlegen neuer Datensätze verfahren werden soll. Außerdem wollen wir bei der Verwaltung der Kunden noch ein paar Anpassungen durchführen, bevor wir die übrigen Ansichten nach einem ähnlichen Schema aufbauen.

Beispieldaten

Als Beispielprojekt verwenden wir weiterhin das Projekt **Bestellverwaltung_SQLite**, das Sie im Download etwa des Beitrags **EDM: 1:n-Beziehungen mit DataGrid** finden.

Aktueller Stand

Der Stand vor dieser Ausgabe enthält ein kleines Ribbon mit drei Schaltflächen, welche die Befehle zum Öffnen der Kundenübersicht, zum Öffnen eines Detailformulars zum Anlegen eines neuen Kunden und zum Löschen des aktuell in der Kundenübersicht markierten Datensatzes anzeigt. An der Schaltfläche **Kunde löschen** haben wir gezeigt, wie Sie in Abhängigkeit des im **Frame**-Objekt angezeigten **Page**-Elements Steuerelemente im Ribbon aktivieren und deaktivieren können.

Die Schaltfläche sollte aktiviert werden, wenn die Kundenübersicht im **Frame**-Element angezeigt wurde und deaktiviert werden, wenn ein anderes **Page**-Element im **Frame**-Element zu sehen war – oder gar keine.

Einen bestehenden Kunden konnten Sie anzeigen, indem Sie erst die Kundenübersicht geöffnet und dann doppelt auf einen der Einträge geklickt haben. Das nun erscheinende Detailformular enthielt auch zwei Schaltflächen namens **Speichern** und **Abbrechen** (siehe Bild 1). Die **Speichern**-Schaltfläche hat die Daten gespeichert und die **Page** geschlossen, die **Abbrechen**-Schaltfläche hat die **Page** geschlossen, ohne zu speichern. In beiden

Bild 1: Übersicht der Kategorien

Fällen wurde danach wieder die Seite mit der Kundenübersicht aufgerufen, wobei der soeben bearbeitete Kunde markiert wurde.

Neue Elemente

In dieser Ausgabe wollen wir der Anwendung weitere Übersichten und Detailansichten hinzufügen. Zunächst einmal wäre da die Produktübersicht mit dem jeweiligen Detailformular zur Anzeige eines Produkts. Diese beiden **Page**-Elemente unterscheiden sich nicht von denen zur Anzeige der Kunden-Übersicht und der Kundendetails.

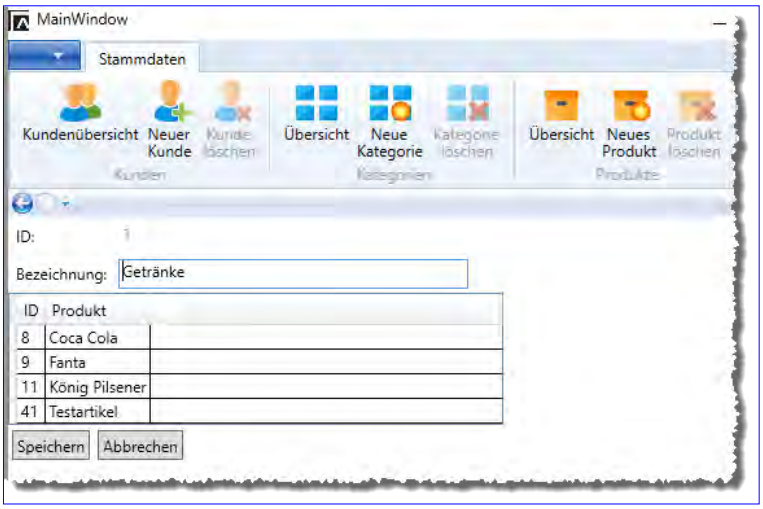


Bild 2: Kategorie-Details mit zugehörigen Produkten

Wir haben diese aber hinzugefügt, weil wir die Produkte noch in einer anderen Ansicht anzeigen möchten, und zwar in einer Kategorie-Detailansicht (mit der natürlich eine passende Kategorien-Übersicht kommt). Die Kategorie-Detailansicht soll nicht nur die Kategorie und die beiden Eigenschaften **ID** und **Bezeichnung** anzeigen, sondern auch die Produkte, die zu der jeweiligen Kategorie gehören (siehe Bild 2).

Von der Übersicht zum Detail und wieder zurück

Wir wollen nun sowohl in der Übersicht der Produkte als auch in der Liste der Produkte in der Kategorie-Detailansicht die Möglichkeit bieten, ein Produkt per Doppelklick in der Detailansicht anzuzeigen. Das ist kein Problem, wir können dabei wie schon bei der Kundenübersicht vorgehen. Was aber geschieht, wenn wir in der Produkt-Detailansicht auf die Schaltfläche **Speichern** oder **Abbrechen** klicken? Bei der Kunden-Detailansicht haben wir einfach die Kundenübersicht angezeigt und den zuletzt geöffneten Kunden markiert.

Nun haben wir aber zwei Produktübersichten (die eigentliche Übersicht und die Übersicht in den Kategorie-Details), von denen aus die Produkt-Detailansicht geöffnet worden sein könnte. Wir haben nun zum Beispiel die folgenden Möglichkeiten:

- Wir fügen in den Konstruktor des **Page**-Elements **Produktdetails.xaml** einen Parameter ein, dem wir den Namen des aufrufenden **Page**-Elements mitgeben und zeigen dieses nach dem Schließen der Produktdetails wieder an.

- Wir ändern die Funktion der beiden Schaltflächen **Speichern** und **Abbrechen**, sodass diese nicht mehr das **Page**-Element **Produktdetails.xaml** ausblenden. Die **Speichern**-Schaltfläche sollte dann nur noch das aktuell angezeigte Produkt speichern, die **Abbrechen**-Schaltfläche könnte dann entfallen. Oder man ersetzt die Abbrechen-Schaltfläche durch eine Verwerfen-Schaltfläche, welche den Zustand vor den Änderungen wieder herstellt. Hier wäre es dann sinnvoll, wenn die **Speichern**-Schaltfläche nur aktiviert wird, wenn es tatsächlich Änderungen gegeben hat.

- Wir nutzen die Eigenschaften des **Frame**-Elements, das ja das Navigieren zum nächsten oder vorherigen Element erlaubt.

Grundsätzlich sollten wir hier auch noch eine Funktion einfügen, die den Benutzer beim Wechsel zu einer anderen Ansicht ohne vorheriges Speichern darauf hinweist, dass eventuell vorgenommene Änderungen noch nicht gespeichert wurden.

Wir entscheiden uns für die folgende Vorgehensweise: Wir nutzen die Navigationsmöglichkeiten des **Frame**-Elements kombiniert mit je zwei Schaltflächen auf jeder Detailansicht, welche die Beschriftungen **Speichern** und **Verwerfen** enthalten.

Drückt der Benutzer eine dieser Schaltflächen, wird das zuvor angezeigte **Page**-Element angezeigt, was das aufrufende Element sein dürfte – im Falle der Produkt-Detailseite also etwa die Produkt-Übersicht oder die Kategorie-Detailseite mit der Liste der verknüpften Produkte.

Sollte der Benutzer die Navigationsschaltflächen zum Zurückspringen nutzen, ohne auf eine der Schaltflächen Speichern oder Verwerfen zu klicken, soll eine Meldung erscheinen, die den Benutzer fragt, ob die Änderungen gespeichert oder verworfen werden sollen. Die Schaltflächen sollen nur aktiviert werden, wenn Änderungen stattgefunden haben, dementsprechend soll die Meldung auch nur in diesem Fall erscheinen.

Markieren des zuletzt bearbeiteten Eintrags

Im bisherigen Stand des Projekts haben wir im Falle des Änderns oder Neuanlegens eines Kunden im **Page**-Element **Kundendetails**.xaml beim Schließen ein Ereignis ausgelöst, welches das Objekt mit dem aktuellen Kunden bereitstellt. Dieses haben wir im übergeordneten **MainWindow** implementiert. So konnten wir, wenn ein Kunde geändert oder hinzugefügt wurde, gleich die Kundenübersicht anzeigen und dort den geänderten oder hinzugefügten Kunden markieren. Dieses Verhalten wollen wir auch beibehalten.

Löschen-Schaltflächen

Auch über die **Löschen**-Schaltflächen sollten wir uns Gedanken machen. Wenn wir für jede Entität nun jeweils eine Schaltfläche zum Anzeigen der Übersicht, zum Anlegen eines neuen Datensatzes und zum Löschen des jeweils markierten Datensatzes hinzufügen, nimmt das erstens Platz im Ribbon weg. Zweitens müssen wir zum Aktivieren und Deaktivieren der Löschen-Schaltflächen mehr und mehr Code zum Code behind-Modul des Fensters **MainWindow.xaml** hinzufügen. Wir entscheiden an dieser Stelle, die Schaltflächen direkt in die jeweiligen **Page**-Elemente zu verlagern, sodass sie direkt an Ort und Stelle aktiviert und deaktiviert werden können.

Hinzufügen-Schaltflächen

Und letztlich nehmen wir auch etwas Komplexität aus dem Code, wenn wir auch eine Schaltfläche zum Hinzufügen eines neuen Elements direkt auf der Page mit der Übersicht anlegen. Auf diese Weise können die Detailansichten der jeweiligen Entitäten zum Hinzufügen auch nur von der entsprechenden Übersicht aus geöffnet werden, was dazu führt, dass wir nach dem Speichern oder Verwerfen der Änderungen am neuen Datensatz auf jeden Fall wieder zu der aufrufenden Übersicht zurückkehren. Gleichermaßen wollen wir aber auch die Möglichkeit bieten, ohne Umweg über die Übersicht einen neuen Datensatz der jeweiligen Entität anzulegen und behalten die entsprechenden Einträge im Ribbon bei.

Zusammenhänge zwischen XAML, C# und den verschiedenen Ansichten und Ereignissen

Auch wenn wir erst wenige Bereiche wie die Kundenübersicht und die Kundendetailansicht zu unserer Lösung hinzugefügt haben, möchten wir einen kleinen Zwischenstopp machen und uns den aktuellen Stand inklusive der oben erwähnten Änderungen ansehen. Dazu haben wir ein paar Übersichten erstellt, welche die einzelnen Abläufe in der Anwendung vorstellen. Diese zeigen wir in den nächsten Abschnitten. Auf Basis der hier beschriebenen Abläufe wollen wir auch die Seiten für weitere Elemente wie die Kategorien und die Produkte aufbauen.

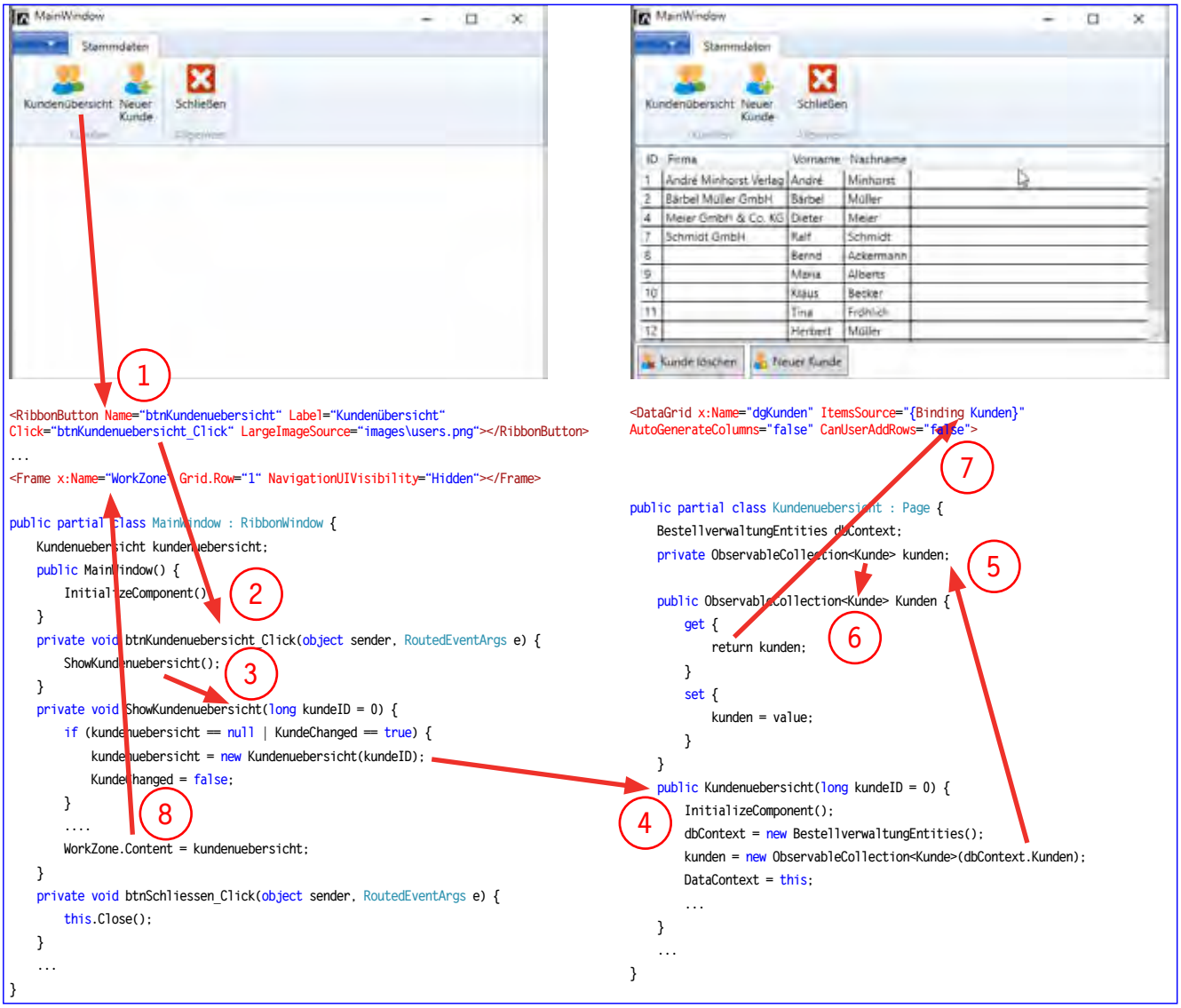


Bild 3: Vorgang beim Öffnen der Übersicht

Anzeigen der Übersicht

Der erste Schritt beim Arbeiten mit den Kunden dürfte die Anzeige der Kundenübersicht sein (siehe Bild 3). Die Anwendung startet mit der Anzeige des nackten Ribbons, das **Frame**-Element im unteren Bereich zeigt noch keine Elemente an. Wenn der Benutzer nun auf die Schaltfläche **cmdKundenuebersicht** klickt, löst er das Ereignis **cmdKundenuebersicht_Click** aus **(1)**.

Dieses haben wir in der Code behind-Klasse mit der Ereignismethode **btnKundenuebersicht_Click** implementiert **(2)**. Die Methode ruft eine weitere Methode namens **ShowKundenuebersicht** auf **(3)**. Dies hat den Hintergrund, dass wir diese Methode noch von anderen Stellen aus aufrufen wollen.

ShowKundenuebersicht prüft hier, ob **kundenuebersicht** bereits eine Kundenübersicht enthält – beispielsweise, weil wir diese zuvor schon einmal angezeigt haben. Das ist hier nicht der Fall, also erstellen wir eine neue Kundenübersicht.

Dadurch wird die Konstruktormethode **Kundenuebersicht** der gleichnamigen Klasse aufgerufen (4). Diese erstellt einen neuen DBContext und füllt die Auflistung **kunden** mit den **Kunde**-Entitäten des Entity Data Models. Außerdem weist sie die **Kunden**-Liste dann der Eigenschaft **this.DataContext** zu, wodurch diese zur Datenquelle für das **Page**-Objekt wird. Durch das Füllen des **ObservableCollection**-Objekts **kunden (5)** wird diese Liste über die öffentliche Eigenschaft **Kunden (6)** auch von außerhalb der Klasse zugreifbar.

So kann nun auch das **DataGrid**-Element der Page **Kundenuebersicht** über das Attribut **ItemsSource** und die Eigenschaft **Kunden** auf die in **kunden** gespeicherte **ObservableCollection** zugreifen **(7)**. Nachdem das **Page**-Element **Kundenuebersicht** nun initialisiert und mit den gewünschten Daten gefüllt ist, kann diese als Inhalt des **Frame**-Elements des Hauptfensters **MainWindow** angegeben werden **(8)**.

Anlegen eines neuen Kunden per Ribbon-Befehl

Das Anlegen eines neuen Kunden wird in Bild 4 schematisch dargestellt.

Wenn der Benutzer auf die Schaltfläche **btnNeuerKunde** im Ribbon klickt, löst er damit die für diese Schaltfläche definierte Ereignismethode **btnNeuerKunde_Click** aus **(1)**. Diese Ereignismethode ist im Code behind-Modul **MainWindow.xaml.cs** definiert und ruft die Methode **NeuerKunde** auf **(2)**. Die Methode erstellt im ersten Schritt eine neue Instanz der Klasse **Kundendetails**. Damit löst sie die Konstruktormethode **Kundendetails** aus, der in diesem Fall kein Parameter übergeben wird, weshalb die Variable **kundeID** den Wert **0** annimmt **(3)**. Die Methode erstellt einen neuen DBContext und weist die aktuelle Klasse als **DataContext** der **Page** zu. Außerdem prüft sie, ob **kundeID** einen Wert ungleich **0** enthält, was hier nicht der Fall ist.

Daher wird im **else**-Teil der Bedingung eine neue Instanz der **Kunde**-Klasse erstellt und in der Variablen **kunde** gespeichert, die dann über die Eigenschaft **Kunde** öffentlich verfügbar ist. Außerdem fügen wir dieses Objekt auch gleich mit der **Add**-Methode der Auflistung **dbContext.Kunden** hinzu **(4)**. Dies bedeutet noch nicht, dass der neue Kunde in der Datenbank gespeichert ist – dies geschieht erst, wenn wir die SaveChanges-Methode aufrufen, was weiter unten der Fall ist. Wenn der Benutzer die Änderung am neuen Kunden verwirft, wird einfach nicht die **SaveChanges**-Methode aufgerufen, wodurch die Lebensdauer des neuen **Kunde**-Objekts einfach gemeinsam mit **dbContext** und den übrigen lokalen Variablen endet.

Außerdem füllt diese Methode noch die Liste **anreden** mit den Elementen der Entitätsliste **Anreden** des Datenbankkontextes **(5)**. Diese steht dann über die Eigenschaft **Anreden** für den Zugriff von den Elementen von **Kundendetails.xaml** bereit **(6)**.

Die Klasse **Kundendetails.xaml.cs** stellt außerdem zwei Ereignisse namens **ItemChanged** und **ItemCancelled** bereit. Diese werden mit den vier Zeilen unter **(7)** deklariert und mit einem Delegaten ergänzt. Damit sind die Arbeiten zum Initialisieren der Seite **Kundendetails** erledigt und wir kehren zum Code behind-Modul **MainWindow.xaml.cs** zurück. Nachdem wir die Objektvariable **kundendetails** mit einem neuen Objekt auf Basis der Klasse **Kundendetails** gefüllt haben, können wir die beiden dort bereitgestellten Ereignisse implementieren. Dazu fügen zunächst entsprechende Ereignishandler hinzu **(8)**. Und schließlich sorgen wir dafür, dass das **Page**-Objekt **kundendetails** im **Frame**-Element **WorkZone** eingeblendet wird **(9)**.

Nun fehlen noch die Schritte, die auf der Page **Kundendetails** ausgeführt werden, um tatsächlich einen neuen Kunden anzulegen. Nach dem Eintragen der Eigenschaften des neuen Kunden in das leere Formular klickt der Benutzer dazu auf die Schaltfläche **btnSpeichern (10)**. Für diese Schaltfläche haben wir die Ereignismethode **btnSpeichern_Click** hinterlegt, die wie unter

EDM: 1:n-Beziehungen mit DataGrid

Unter Access haben wir 1:n-Beziehungen einfach in einem Haupt- und einem Unterformular abgebildet, wobei wir beiden einfach die Datenquellen und gebundenen Steuerelemente zugewiesen haben – den Rest hat Access automatisch erledigt. Unter C# und WPF ist das ein wenig mehr Arbeit, aber nach der Lektüre dieses Artikels haben Sie das Wissen, das für die Anzeige zweier per 1:n-Beziehung verknüpfter Tabellen in einem Fenster beziehungsweise einer Seite und einem DataGrid als Unterformular-Ersatz nötig ist.

Beispieldaten

Als Beispiel wollen wir uns die Kategorien und die damit verknüpften Produkte ansehen. Dabei sollen die beiden Felder einer Kategorie im Fenster/in der Seite selbst angezeigt werden, die dazugehörigen Produkte in einem DataGrid unter den Kategoriedaten. Wir verwenden die SQLite-Datenbank **Bestellverwaltung.db** und ein daraus abgeleitetes Entity Data Model namens **BestellverwaltungEntities.edmx** als Datenquelle. Die verwendeten Auflistungen beziehungsweise Entitäten heißen **Kategorien**, **Kategorie**, **Produkte** und **Produkt**.

Einbau in die Bestellverwaltung

In einigen weiteren Beiträgen verwenden wir die Bestellverwaltung als Beispielanwendung. Diese verwendet im Office-Stil ein Ribbon zur Auswahl der verschiedenen Bereiche und Funktionen. Die einzelnen Bereiche etwa zur Anzeige einer Kundenliste oder einer Kundendetailansicht auf Seiten (**Page**) statt auf eigenen Fenstern (**Window**) erstellt und je nach angeklickter Ribbon-Schaltfläche in einem Frame-Element eingeblendet – also etwa wie die Unterformulare in einem Unterformular-Steuerelement in Access. Wir fügen also in diesem Artikel weitere **Page**-Elemente hinzu, die wir dann nach dem Anklicken der ebenfalls noch anzulegenden Ribbon-Schaltflächen angezeigt werden sollen.

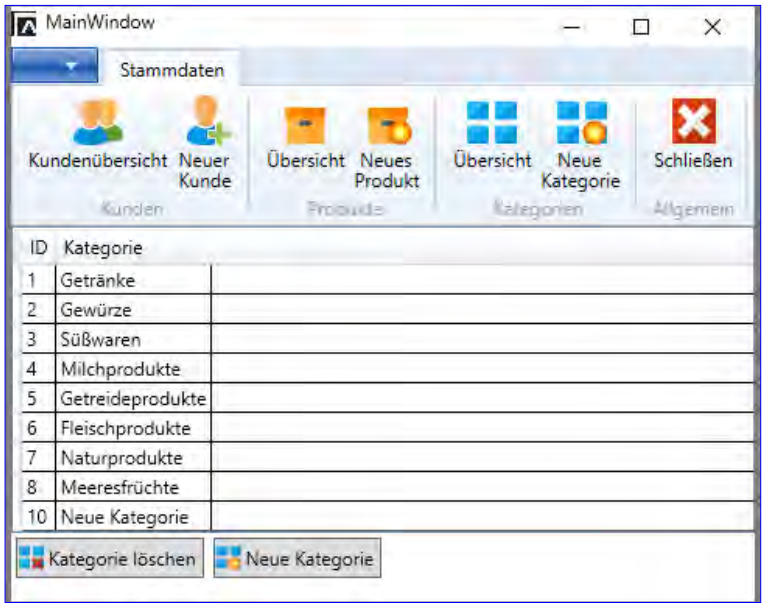


Bild 1: Übersicht der Kategorien

Ribbon-Einträge

Als Erstes legen wir die drei Ribbon-Einträge an, die Sie in Bild 1 sehen. Die Elemente für diese Ribbon-Gruppe sehen wie folgt aus:

```
<RibbonGroup Header="Kategorien">                                                    //MainWindow.xaml
    <RibbonButton Name="btnKategorieuebersicht" Label="Übersicht" Click="btnKategorieuebersicht_Click"
        LargeImageSource="images/elements4.png"></RibbonButton>
```

```
    <RibbonButton Name="btnNeueKategorie" Label="Neue Kategorie" Click="btnNeueKategorie_Click"
        LargeImageSource="images/elements4_new.png"></RibbonButton>
    <RibbonButton Name="btnKategorieLoeschen" Label="Kategorie löschen" Click="btnKategorieLoeschen_Click"
        LargeImageSource="images/elements4_delete.png"></RibbonButton>
</RibbonGroup>
```

Wir haben für jedes Element ein Bild hinterlegt, dass wir gleichzeitig zum Ordner **images** des Projekts hinzugefügt haben.

Übersichtsseite für die Kategorien

Die Kategorien werden in der Übersichtsseite **KategorieUebersicht.xaml** in einem **DataGrid**-Element aufgelistet. Das Füllen dieses **DataGrid**-Elements erfolgt genauso, wie wir es bereits im Artikel **EDM: Kunden verwalten mit Ribbon** für die Anzeige der Kunden realisiert und im Artikel **Bestellveraltung planen** angepasst haben – nur, dass wir diesmal auf die Entitätsliste der Kategorien statt der Kunden zugreifen und nur die Felder **ID** und **Bezeichnung** anzeigen.

Detailseite für die Kategorien

Der interessante Teil folgt nun, nämlich die Detailseite für eine Kategorie. Eine Kategorie enthält zwar nur die beiden Felder **ID** und **Bezeichnung**, was sich genau so leicht abbilden lässt wie in der Detailansicht für Kunden (**Kundendetails.xaml**) – nur mit weniger Feldern. Allerdings wollen wir ja zu jeder Kategorie auch noch die Liste der Produkte anzeigen, die der jeweiligen Kategorie zugeordnet sind! Und das wollen wir wiederum mit einem **DataGrid**-Element erledigen.

Imgrunde brauchen wir also eine Kombination der bereits einmal gezeigten Detailansicht für die Kunden (diesmal für die Kategorien) und einer Übersicht für die Produkte. Der Unterschied bei der Übersichtsseite diesmal ist, dass wir nicht mehr einfach alle Elemente der Produkte-Liste anzeigen können, sondern nur noch diejenigen Elemente, die der aktuellen Kategorie zugeordnet sind. Die Definition der Seite mit den Kategoriedetails und der Produktliste startet mit ein paar **Page.Resources**-Elementen, welche globale Attribute der verschiedenen Steuerelemente festlegen und die wir hier ebenso gekürzt haben wie die Definition des Grids:

```
<Page x:Class="Bestellverwaltung.Kategoriedetails" ...Title="Kategoriedetails">                    //Kategoriedetails.xaml
    <Page.Resources>...</Page.Resources>
    <Grid>
        <Grid.ColumnDefinitions>...</Grid.ColumnDefinitions>
        <Grid.RowDefinitions>...</Grid.RowDefinitions>
```

Danach folgt das Label für die **ID** der Kategorie und die **TextBox**, die wir an das Feld **kategorie.ID** binden:

```
        <Label Content="ID:" Grid.Column="0" />
        <TextBox x:Name="txtID" Grid.Column="1" HorizontalAlignment="Left" Text="{Binding kategorie.ID, Mode=TwoWay,
            ValidatesOnExceptions=true}" Width="50" IsEnabled="False" BorderBrush="Transparent" />
```

Die Bezeichnung erhält ebenfalls ein Label und eine Bindung an das Feld **kategorie.Bezeichnung** – beide landen in jeweils einer Spalte, genau wie die Steuerelemente für die ID:

```
        <Label Content="Bezeichnung:" Grid.Column="0" Grid.Row="1" />
        <TextBox x:Name="txtBezeichnung" Grid.Column="1" HorizontalAlignment="Stretch" Grid.Row="1"
            Text="{Binding kategorie.Bezeichnung, Mode=TwoWay, ValidatesOnDataErrors=true}" />
```

Schließlich folgt das **DataGrid**-Element namens **dgProdukte**, das in der folgenden Zeile landet und sich über zwei Spalten erstrecken soll (**Grid.ColumnSpan="2"**). Als **ItemsSource** für das **DataGrid** legen wir das Element **Produkte** fest. Das automatische Generieren der Spalten sowie das Anzeigen einer leeren Spalte zum Hinzufügen von Elementen deaktivieren wir:

```
        <DataGrid x:Name="dgProdukte" Grid.Column="0" Grid.Row="2" Grid.ColumnSpan="2" ItemsSource="{Binding Produkte}"
                AutoGenerateColumns="false" CanUserAddRows="False">
```

Die beiden Spalten des **DataGrid**-Elements binden wir an die Felder **ID** und **Bezeichnung**:

```
            <DataGrid.Columns>
                <DataGridTextColumn Binding="{Binding Path=ID}" Header="ID"  />
                <DataGridTextColumn Binding="{Binding Path=Bezeichnung}" Header="Produkt" />
            </DataGrid.Columns>
```

Außerdem wollen wir, wie schon bei den übrigen Übersichten, ein Öffnen der Produktdetails per Doppelklick erlauben und fügen dazu einen **EventSetter** hinzu:

```
            <DataGrid.Resources>
                <Style TargetType="DataGridRow">
                    <EventSetter Event="MouseDoubleClick" Handler="Row_DoubleClick"/>
                </Style>
            </DataGrid.Resources>
        </DataGrid>
```

Schließlich folgen noch die beiden Schaltflächen zum Speichern und Verwerfen der aktuellen Änderungen:

```
        <StackPanel Orientation="Horizontal" Grid.Row="6" Grid.ColumnSpan="4" >
            <Button x:Name="btnSpeichern" Margin="3" Padding="3" Click="btnSpeichern_Click" Height="23"
                Content="Speichern"></Button>
            <Button x:Name="btnVerwerfen" Margin="3" Padding="3" Click="btnVerwerfen_Click" Height="23"
                Content="Verwerfen"></Button>
        </StackPanel>
    </Grid>
</Page>
```

Der Entwurf sieht in der XAML-Ansicht nun wie in Bild 2 aus. Im Entwurf haben wir nun schon einige Bindungen gesehen, die wir nun in der Code behind-Klasse bereitstellen wollen. Diese Klasse enthält zunächst eine Variable für das **DBContext**-

Element, über das wir auf das Entity Data Model zugreifen (**dbContext**). Außerden finden wir hier die Definition für die Objekte, an die wir die Elemente der XAML-Seite binden wollen. Die erste ist das **Kategorie**-Objekt, das wir in der öffentlichen Variablen **kategorie** speichern.

Die zweite ist das **ObservableCollection**-Objekt **Produkte**, deren Inhalt wie in der als privat deklarierten Variablen **produkte** speichern und über entsprechende Eigenschaften für den Zugriff von außen bereitstellen:

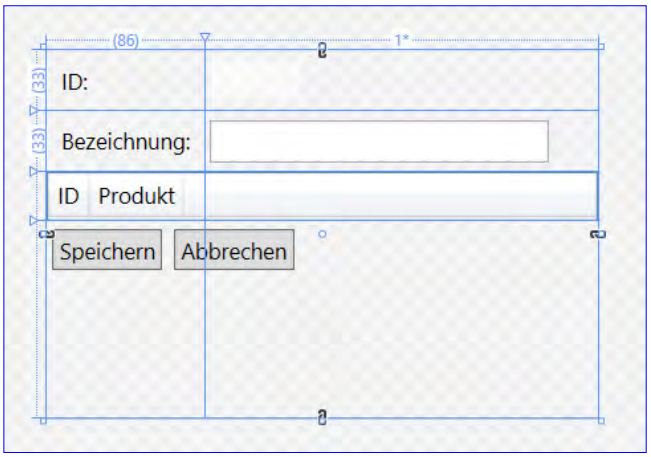


Bild 2: Entwurf der Kategoriedetail-Seite

```
public partial class Kategoriedetails : Page {
//Kategoriedetails.xaml.cs
    private BestellverwaltungEntities dbContext;
    public Kategorie kategorie { get; set; }
    private ObservableCollection<Produkt> produkte;
    public ObservableCollection<Produkt> Produkte {
        get {
            return produkte;
        }
        set {
            produkte = value;
        }
    }
    ...
}
```

Die Konstruktor-Methode **Kategoriedetails**, die beim Erstellen eines Objekts auf Basis unserer Klasse aufgerufen wird, erwartet als Parameter den Primärschlüsselwert der ID einer Kategorie. Wird dieser nicht übergeben, setzt die Methode ihn auf **0** fest – dies bedeutet, dass ein neues, leeres **Kategorie**-Element angezeigt werden soll.

Die Methode erstellt ein neues Objekt auf Basis der Klasse **BestellverwaltungEntities** und stellt den **DataContext** auf die Code behind-Klasse (**this**) ein, damit die XAML-Elemente an die per Eigenschaft zugängig gemachten Elemente der Code behind-Klasse zugreifen können. Ist die mit dem Parameter **kategorieID** übergebene **ID** der Kategorie nicht gleich **0**, sucht die **Find**-Methode in der **Kategorien**-Auflistung nach dem entsprechenden Element. Anderenfalls wird ein neues Kategorie-Element angelegt.

```
public Kategoriedetails(long kategorieID = 0) {                    //Kategoriedetails.xaml.cs
    InitializeComponent();
    dbContext = new BestellverwaltungEntities();
    DataContext = this;
```

```
    if (kategorieID != 0) {
        kategorie = dbContext.Kategorien.Find(kategorieID);
    }
    else {
        kategorie = new Kategorie();
        dbContext.Kategorien.Add(kategorie);
    }
    produkte = new ObservableCollection<Produkt>(dbContext.Produkte.Where(d => d.KategorieID == kategorie.ID));
}
```

Danach folgt der interessante Schritte: Wir füllen das DataGrid, und zwar nur mit den Entitäten, die einem bestimmten Suchkriterium entsprechen.

Dieses soll nur die Entitäten liefern, deren Feld **KategorieID** dem Wert des Feldes **ID** der angezeigten **Kategorie**-Entität entspricht. Dazu verwenden wir einen entsprechenden Linq-Ausdruck (siehe auch Artikel **LINQ to Entities: Daten abfragen**):

```
produkte = new ObservableCollection
<Produkt>(dbContext.Produkte.Where
(d=>d.KategorieID==kategorie.ID));
```

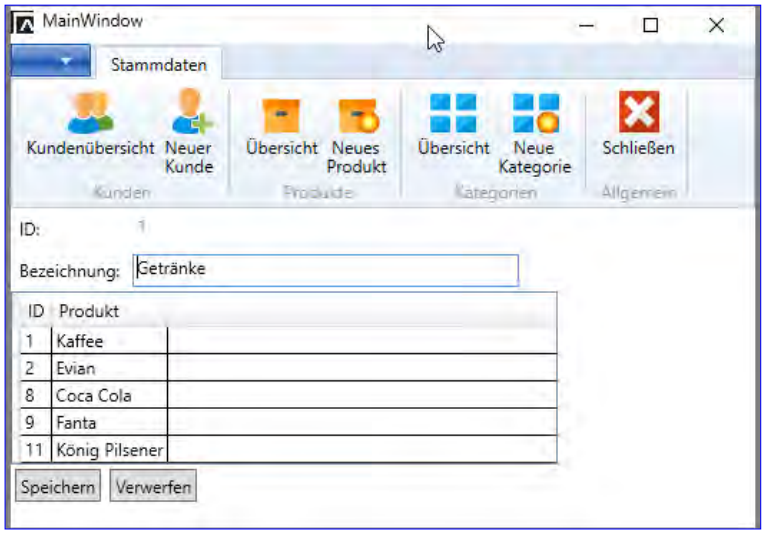


Bild 3: Anzeige von Kategorien und Produkten

Das Ergebnis sieht dann wie in Bild 3 aus.

Fokus auf Bezeichnung

Damit gleich beim Öffnen der Seite der Fokus auf das Textfeld zur Eingabe der Bezeichnung gelegt wird, geben Sie für das **Page**-Objekt den Wert **Page_Loaded** für das Attribut **Loaded** an:

```
<Page x:Class="Bestellverwaltung.Kategoriedetails" ... Loaded="Page_Loaded">                    //Kategoriedetails.xaml
```

Dies soll die folgende Ereignismethode auslösen, die lediglich den Fokus auf das Feld **txtBezeichnung** legt:

```
private void Page_Loaded(object sender, RoutedEventArgs e) {                                    //Kategoriedetails.xaml.cs
    txtBezeichnung.Focus();
}
```

Speichern der Kategorie

Ein Mausklick auf die Schaltfläche **Speichern** löst die folgende Ereignismethode aus. Diese ruft einfach eine weitere Methode namens **Speichern** auf:

```
private void btnSpeichern_Click(object sender, RoutedEventArgs e) {                    //Kategoriedetails.xaml.cs
    Speichern();
}
```

Die Methode **Speichern** speichert die aktuelle **Kategorie**-Entität mit der Methode **SaveChanges**.

Außerdem ruft sie noch das Ereignis **ItemChanged** auf und übergibt ihm eine neue Instanz der Klasse **KategorieEventArgs** mit der aktuellen Kategorie als Konstruktor-Parameter:

```
private void Speichern() {                                                             //Kategoriedetails.xaml.cs
    dbContext.SaveChanges();
    ItemChanged(this, new KategorieEventArgs(kategorie));
}
```

Die Klasse **KategorieEventArgs** implementiert die Schnittstellt **EventArgs**. Dieses bietet die Möglichkeit, die beim Auslösen des Ereignisses (hier **ItemChanged**) notwendigen Daten aufzunehmen und an die Implementierung des Ereignisses zu übergeben.

In diesem Fall soll dies lediglich das geänderte **Kategorie**-Objekt sein:

```
public class KategorieEventArgs : EventArgs {                                          //KategorieEventArgs.cs
    public KategorieEventArgs(Kategorie kategorie) {
        Kategorie = kategorie;
    }
    public Kategorie Kategorie { get; set; }
}
```

Das Ereignisse definieren wir wie folgt und legen den entsprechenden Delegaten fest:

```
public delegate void EventHandlerItemChanged(object sender, KategorieEventArgs e);    //Kategoriedetails.xaml.cs
public event EventHandlerItemChanged ItemChanged;
```

Den Fall des Abbrechens behandelt diese Methode:

```
private void btnAbbrechen_Click(object sender, RoutedEventArgs e) {                    //Kategorieudetails.xaml.cs
    Cancelled(this, null);
}
```

Bei **Cancelled** handelt es sich wieder um ein Ereignis, das samt Delegat wie folgt deklariert wird:

```
public delegate void EventHandlerCancelled(object sender, EventArgs e);                //Kategoriedetails.xaml.cs
public event EventHandlerCancelled Cancelled;
```

Tipps und Tricks zu Visual Studio

Wer von Access zu Visual Studio wechselt, stößt immer wieder auf neue Herausforderungen in dieser neuen, im Vergleich zur Access-Entwicklungsumgebung viel komplexeren Welt. In der heutigen Ausgabe der Reihe Tipps und Tricks zu Visual Studio schauen wir uns einmal an, wie Sie ein Projekt umbenennen. Unter Access haben Sie dazu die Datei und gegebenenfalls noch das VBA-Projekt umbenannt. Unter Visual Studio ist eine ganze Menge mehr zu beachten: Allein, das wir nicht nur eine einzige Datei haben, sondern ganze Ordner voller Dateien und uns auch noch mit zu erstellenden .exe-Dateien und Dingen wie Namespaces herumschlagen müssen, macht die Aufgabe recht komplex ...

Komplettes Projekt umbenennen

Während der Erstellung der Beispielprojekte für dieses Magazin ist es recht häufig passiert, dass ein Projekt nicht gleich den richtigen Namen hatte. Mit dem Projektnamen legt man beim Erstellen eines Projekts natürlich auch gleich noch eine Menge weiterer Benennungen fest, zum Beispiel verschiedene Ordner und Pfade, den Namespace et cetera. Also schauen wir uns einmal an, welche Schritte nötig sind, um ein solches Projekt umzubenennen. Wir sind es zuerst einmal naiv angegangen und haben einfach den Ordner, in dem sich alle Projektdateien befinden, die **.sln**-Datei und den Ordner mit den weiteren Projektdateien in **Bestellverwaltung** umbenannt (siehe Bild 1). Den Rest werden wir schon in Visual Studio ändern können, haben wir uns gedacht.

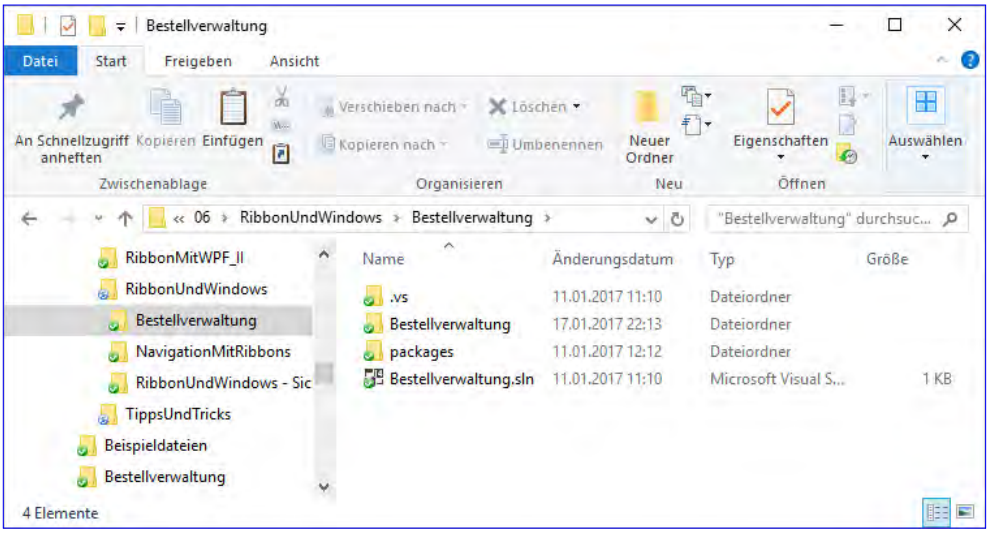


Bild 1: Projektordner mit **.sln**-Datei und Unterordnern

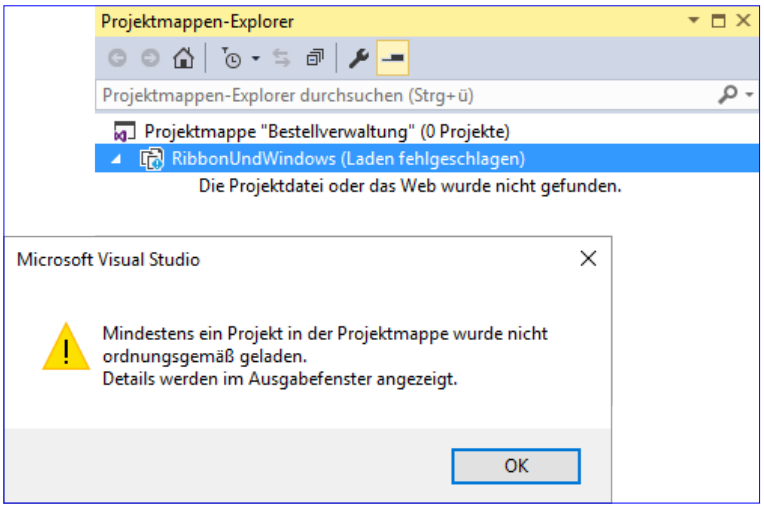


Bild 2: Probleme beim Laden eines Projekts

Aber weit gefehlt: Das Öffnen der **.sln**-Datei per Doppelklick lieferte die Meldung aus Bild 2 und anschließend einen Projektmappen-Explorer mit dem Hinweis, dass das Laden fehlgeschlagen ist. Was nun?

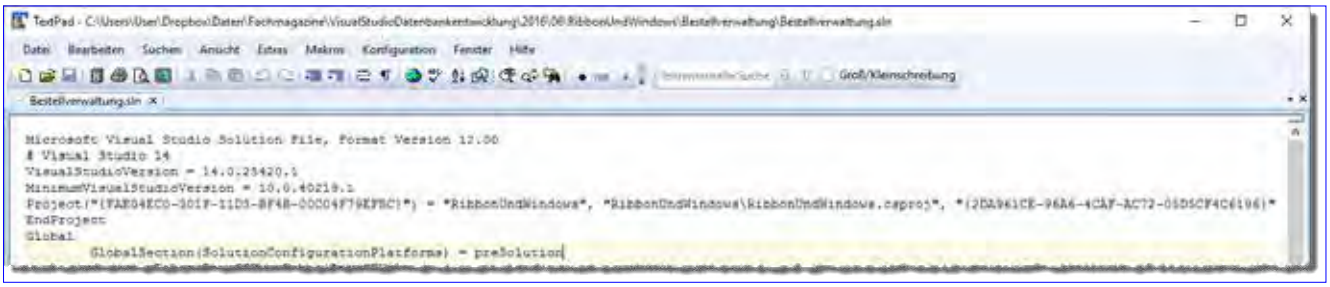


Bild 3: Falscher Pfad in der **.sln**-Datei

Wenn das Laden fehltschlägt, liegt das meist daran, dass die gewünschte Datei nicht an Ort und Stelle ist, was uns ein Blick in das Ausgabefenster dann auch bestätigte.

Da Visual Studio keine Möglichkeit bot, die **.sln**-Datei zu öffnen, haben wir diese also in einem handelsüblichen Editor angezeigt. Und hier tauchten dann auch gleich einige Erwähnungen des vorherigen Projektnamens **RibbonsUndWindows** auf (siehe Bild 3). Tauschen wir diesen also an den relevanten Stellen gegen **Bestellverwaltung** aus, sieht die betroffene Zeile wie folgt aus (auch den Projektnamen haben wir gleich angepasst):

```
Project("{FAE04EC0-301F-11D3-BF4B-00C04F79EFBC}") = "Be-
stellverwaltung", "Bestellverwaltung\Bestellverwaltung.
csproj", "{2DA961CE-96A6-4CAF-AC72-05D5CF4C6196}"
```

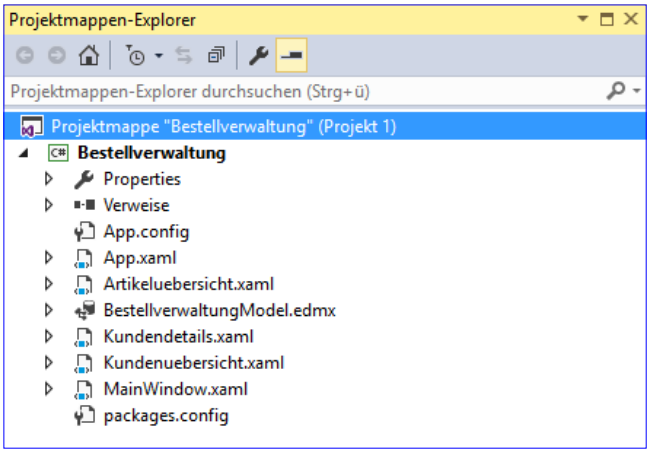


Bild 4: Erfolgreiches Laden des Projekts

Das der nächste Versuch, das Projekt über die **.sln**-Datei in Visual Studio ebenfalls fehlschlägt, hätten wir vorausahnen können: Wir haben ja in der obigen Zeile auch gleich den Namen der **.csproj**-Datei auf **Bestellverwaltung.csproj** geändert, dies aber noch nicht im Windows Explorer nachgeholt. Gegegebenenfalls müssen Sie auch noch andere Dateien umbenennen, deren Name den alten Projektnamen enthält.

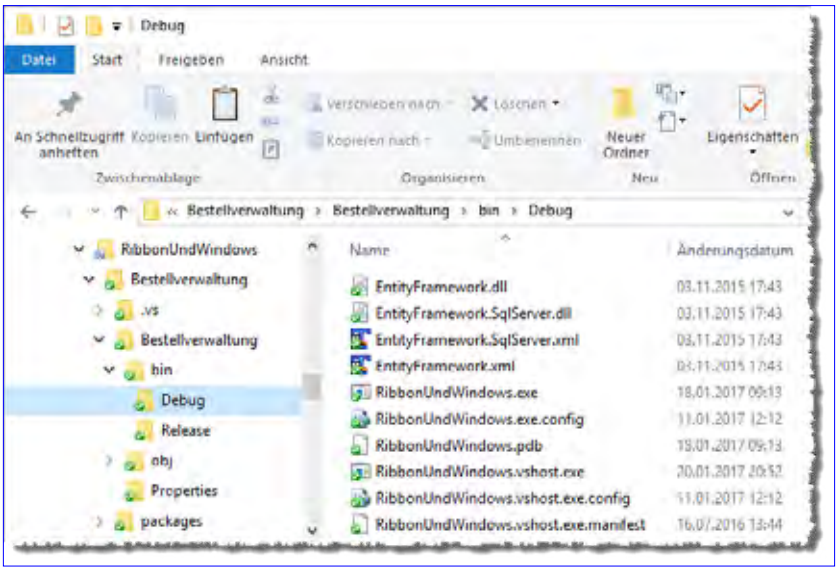


Bild 5: Die erstellten Dateien enthalten noch den alten Projektnamen

Erste wenn wir auch noch diese Datei im Windows Explorer umbenennen, lädt Visual Studio das komplette Projekt (siehe Bild 4). Dies dauert zwar ein paar Sekunden, aber vielleicht hat Visual Studio ja direkt noch ein paar Änderungen bezüglich des Projektnamens im Code vorgenommen.

Der erste Test zeigt, dass das Projekt läuft, aber ein Blick in den Code offenbart, dass

tatsächlich wohl nur die Ordner, die beiden Dateien mit der Endung **.sln** und **.csproj** sowie das Projekt selbst umbenannt wurden (siehe Bild 5). Der Namespace hat seinen Namen noch behalten.

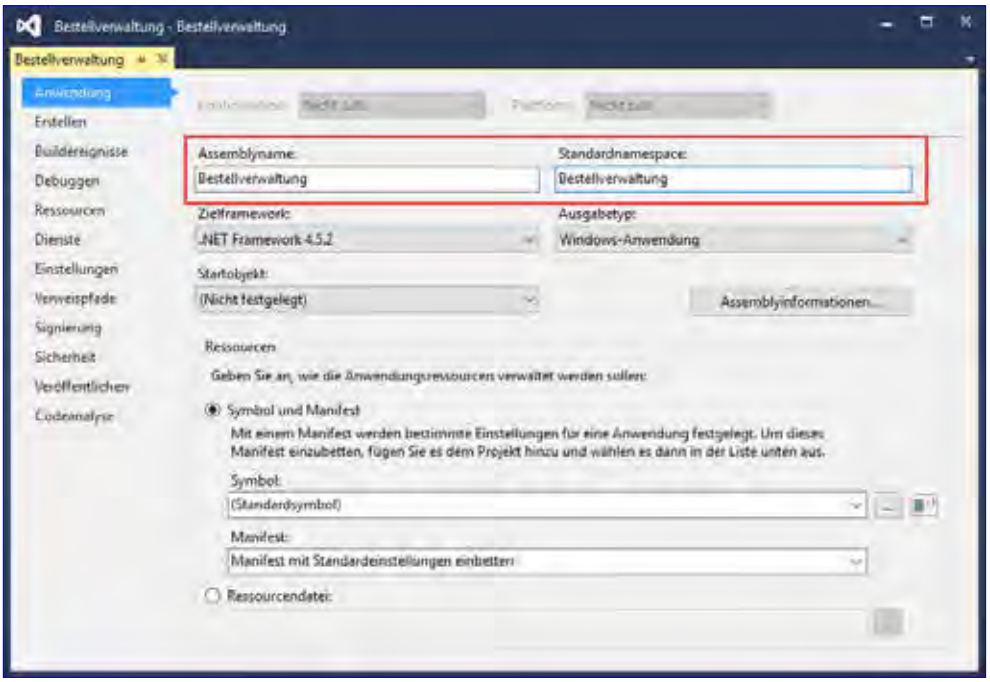


Bild 6: Ändern des Assemblynamens und des Standardnamespaces

Außerdem zeigt ein Blick in den Ordner **bin\Debug**, das auch die erstellten Dateien, also etwa die **.exe**-Datei, noch den alten Namen enthalten. Kümmern wir uns zunächst einmal um diese Dateien. Dazu brauchen Sie nur die Eigenschaften des Projekts zu öffnen, indem Sie im Projektmappen-Explorer doppelt auf den Eintrag **Properties** unterhalb des Projektnamens klicken. Hier stellen Sie direkt auf der ersten Seite **Anwendung** den Wert der Eigenschaft **Assemblyname** auf **Bestellverwaltung** ein (siehe Bild 6). Ein Test zeigt, dass die erstellten Dateien nun mit dem Namen **Bestellverwaltung** beginnen – also **Bestellverwaltung.exe** et cetera. Da liegt nun die Versuchung nahe, auch gleich die Eigenschaft **Standardnamespace** im gleichen Dialog auf **Bestellverwaltung** zu ändern. Also probieren wir dies aus und erstellen das Projekt erneut.

Nun stoßen wir auf Probleme: Zwar hat diese Änderung nicht den Namespace in den durch den Benutzer erstellten geändert. Aber es gibt ja gegebenenfalls (und speziell in den Projekten, die wir in den Beispielen der aktuellen Ausgaben

Die Leseprobe dieses Artikels ist hier zu Ende.
Willst Du mehr?
ZUM SHOP
2.400 Seiten Know-how rund um die Datenbankentwicklung mit .NET
KOMPENDIUM
DATENBANK ENTWICKLER
KOMPENDIUM
André Minhorst Verlag
Spare 20 EUR mit Code dbew20

AUSGABE 02/2017

DATENBANK ENTWICKLER

MAGAZIN FÜR DIE DATENBANKENTWICKLUNG MIT VISUAL STUDIO FÜR DESKTOP, WEB UND CO.

TOP-THEMEN:

C#-GRUNDLAGEN	Datumsfunktionen	SEITE 3
WPF-BASICS	DataGrid im Detail	SEITE 15
INTERAKTIV	DHL-Etiketten per Webservice	SEITE 22
LÖSUNGEN	PDFs zusammenführen	SEITE 43
LÖSUNGEN	EDM: Bestellungen und Bestellpositionen	SEITE 51

Datumsfunktionen

Ohne Datumsfunktionen kommt man in keiner Programmiersprache aus. Das ist unter C# natürlich nicht anders. Wer allerdings von Access/VBA kommt, muss sich ein wenig umstellen: Einzelne Funktionen wie Date() oder Time(), mit denen Sie beispielsweise das aktuelle Datum oder die aktuelle Uhrzeit ermittelt haben, suchen Sie hier vergeblich. Stattdessen nutzen Sie verschiedene Eigenschaften des Objekts DateTime. Der vorliegende Artikel zeigt die wichtigsten Elemente der C#-Programmiersprache zum Thema Datum und erläutert, wie das Datum hier im Hintergrund behandelt wird.

Aktuelles Datum und aktuelle Uhrzeit ausgeben

Als erstes Beispiel wollen wir das aktuelle Datum und die aktuelle Uhrzeit in zwei Textfeldern eines WPF-Fensters ausgeben. Die Steuerelemente dafür definieren wir wie folgt:

```
<Grid>
    <Grid.RowDefinitions>
        <RowDefinition Height="Auto"></RowDefinition>
        <RowDefinition Height="Auto"></RowDefinition>
    </Grid.RowDefinitions>
    <Grid.ColumnDefinitions>
        <ColumnDefinition Width="Auto"></ColumnDefinition>
        <ColumnDefinition Width="Auto"></ColumnDefinition>
        <ColumnDefinition Width="*"></ColumnDefinition>
    </Grid.ColumnDefinitions>
    <Label Content="Aktuelles Datum (DateTime.Now.ToShortDateString):"></Label>
    <TextBox x:Name="txtAktuellesDatum" Grid.Column="1"></TextBox>
    <Button x:Name="btnAktuellesDatum" Content="Aktualisieren" Grid.Column="2" Click="btnAktuellesDatum_Click"></Button>
    <Label Content="Aktuelle Zeit (DateTime.Now.ToShortTimeString):" Grid.Row="1"></Label>
    <TextBox x:Name="txtAktuelleZeit" Grid.Row="1" Grid.Column="1"></TextBox>
    <Button x:Name="btnAktuelleZeit" Content="Aktualisieren" Grid.Row="1" Grid.Column="2"
        Click="btnAktuelleZeit_Click"></Button>
</Grid>
```

Dies ist das Grundgerüst, weitere Elemente rüsten wir im Laufe dieses Artikels nach. In diesem Fall haben wir ein Grid aufgebaut, dass drei Spalten enthält – eines mit der Bezeichnung, eines mit dem per C# ermittelten Inhalt und eines mit einer Schaltfläche, um den Inhalt des Textfeldes zu aktualisieren oder andere Funktionen auszuführen.

Das obere Textfeld soll das aktuelle Datum anzeigen, das untere die aktuelle Uhrzeit. Die Beschriftungen verraten bereits die dort verwendeten C#-Funktionen. In der Konstruktormethode **MainWindow()** füllen wir die Textfelder gleich beim Anzeigen des Fensters:

```
public MainWindow() {
    InitializeComponent();
    txtAktuellesDatum.Text = DateTime.Now.ToShortDateString();
    txtAktuelleZeit.Text = DateTime.Now.ToLongTimeString();
}
```

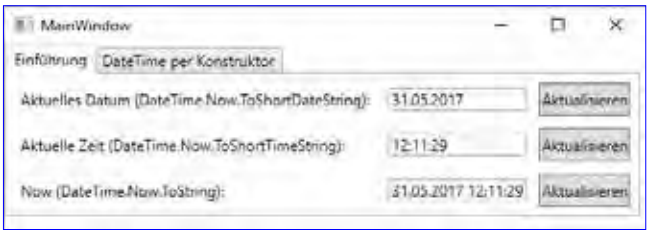


Bild 1: Ausgabe in Textfeldern

Hier sehen Sie, dass wir das aktuelle Datum mit dem Ausdruck **DateTime.Now.ToShortDateString()** ermitteln. Dabei ist **DateTime** das zentrale Objekt, wenn es sich um Datum- und Zeitangaben dreht. Nach diesem einführenden Beispiel schauen wir uns an, was die **DateTime**-Klasse alles bietet. Das Ergebnis sieht wie in Bild 1 aus.

Die DateTime-Klasse

Während wir unter VBA einfach Funktionen wie **Date()** oder **Time()** nutzen konnten, um die aktuelle Uhrzeit zu ermitteln, benötigen wir unter C# natürlich die Methoden eines Objekts für diesen Zweck – beziehungsweise die statischen Methoden einer Klasse. In diesem Fall nutzen wir die Methode **Now** der Klasse **DateTime**, um auf das aktuelle Datum und die aktuelle Uhrzeit zuzugreifen, also **DateTime.Now**. Allerdings reicht dies noch nicht aus, denn wir benötigen noch ein Ausgabeformat. **DateTime.Now** liefert nämlich eine Struktur des Typs **DateTime** zurück, deren Inhalt so nicht direkt ausgegeben werden kann.

Dazu nutzen wir dann wie im obigen Beispiel etwa die Methoden **ToShortDateString()** oder **ToLongTimeString()**.

Datentyp für das Datum und Ticks

Der Datentyp zum Speichern von Datumswerten heißt unter C# **DateTime**. Genaugenommen handelt es sich um eine Struktur, mit der Sie Zeitangaben vom 1.1.0001 bis zum 31.12.9999 darstellen können. Intern wird das Datum als **long**-Wert in der Einheit **Tick** gespeichert. Ein Tick entspricht hundert Nanosekunden – das sollte für die meisten Anwendungsfälle ausreichend genau sein. Damit steht die interne Speicherung im Gegensatz zu der etwa unter VBA: Hier wird ein Datum als Fließkommazahl gespeichert, wobei der Teil vor dem Komma dem Datum und der Teil nach dem Komma der Uhrzeit entspricht.

Neben den Ticks als **long**-Wert enthält **DateTime** auch noch die **Kind**-Eigenschaft, welche speichert, ob es sich um ein lokales Datum, ein UTC-Datum (mehr dazu weiter unten) oder ein unspezifisches Datum handelt.

Die verschiedenen Darstellungen wie etwa langes Datum, kurzes Datum, lange Uhrzeit, kurze Uhrzeit oder auch Kombinationen sind lediglich Umrechnungen der gespeicherten Anzahl Ticks in einen Datumswert und die Ausgabe als String.

Statische Funktionen der DateTime-Klasse

Die **DateTime**-Klasse liefert eine Reihe Funktionen rund um Datumsangaben, die Sie kennen sollten:

Compare: Erwartet zwei **DateTime**-Objekte und gibt folgende Werte zurück: **-1**, wenn das erste Datum kleiner als das zweite Datum ist, **0**, wenn beide Daten genau gleich sind und **1**, wenn das zweite Datum kleiner als das erste Datum ist.

DaysInMonth: Erwartet die Angabe von Jahr und Monat als Parameter und liefert die Anzahl der Tage dieses Monats zurück. Beispiel: **DaysInMonth(2017, 1)** liefert **31**.

Equals: Gibt für die beiden als Parameter übergebenen **DateTime**-Objekte an, ob diese gleich sind. Beispiel: **DateTime.Equals(new DateTime(2017, 1, 1), new DateTime(2017, 1, 1)).ToString()** liefert **true**.

FromBinary: Liest ein Datum von einer binären Repräsentation eines Datums im **long**-Format wieder in ein Datum ein. Dazu benötigen Sie zunächst die binäre Form, welche wir im Vorgriff auf die Methoden von Objekten auf Basis der Klasse **DateTime** mit der Methode **ToBinary** erstellen. Wir erstellen hier ein Datum mit **Now** und speichern seine mit **ToBinary()** erstellte binäre Repräsentation in der Variablen **lngDat**. Dann geben wir das Originaldatum, die binäre Variante und das aus dieser Variante zurückverwandelte **DateTime**-Objekt aus:

```
DateTime dat = DateTime.Now;
long lngDat = dat.ToBinary();
Console.WriteLine("Orig. Datum: {0}", dat.ToString());
Console.WriteLine("Datum binär: {0}", lngDat.ToString());
Console.WriteLine("FromBinary:  {0}", DateTime.FromBinary(lngDat).ToString());
```

FromFileTime: Wie FromBinary, allerdings wird hier eine mit der Methode ToFileTime erstellte Anzahl von Ticks wieder zurückverwandelt.

FromFileTimeUTC: Wie **FromFileTime**, nur dass die mit der Methode **ToFileTimeUTC** erstellte Anzahl von Ticks für eine UTC-Zeit wieder zurückverwandelt wird.

FromOADate: Wandelt ein Datum, dass unter VBA/OLE erstellt wurde (also die Repräsentation durch eine Fließkommazahl, deren Vorkommastellen das Datum und deren Dezimalstellen die Uhrzeit markieren), in ein **DateTime**-Objekt um.

IsLeapYear: Gibt an, ob es sich bei dem als Parameter übergebenen Jahr um ein Schaltjahr handelt. Beispiel: **DateTime.IsLeapYear(2016).ToString()** liefert **true**.

MaxValue: Liefert den größtmöglichen Wert eines **DateTime**-Objekts. Beispiel: **DateTime.MaxValue().ToString()** liefert **31.12.9999 23:59:59**.

MinValue: Liefert den kleinstmöglichen Wert eines **DateTime**-Objekts. Beispiel: **DateTime.MinValue().ToString()** liefert **01.01.0001 00:00:00**.

Now: Liefert ein **DateTime**-Objekt mit dem aktuellen Datum und der aktuellen Uhrzeit. Das **DateTime**-Objekt offeriert weitere Methoden (siehe unten).

Parse: Erwartet eine Zeichenkette mit einem Datum, welches dann in ein **DateTime**-Objekt konvertiert wird, zum Beispiel **DateTime.Parse("1.2.2017 10:11:12").ToString()**.

ParseExact: Wie Parse, nur dass hier noch das genaue Format vorgegeben werden muss: **DateTime.ParseExact("01.02.2017 10:11:12", "dd.MM.yyyy hh:mm:ss", null).ToString();**

ReferenceEquals: Vergleicht zwei Instanzen und gibt an, ob diese gleich sind. Beispiel: **DateTime.ReferenceEquals(new DateTime(2017,1,1), new DateTime(2017,1,1)).ToString()** liefert **false**, da es sich ja um zwei unabhängig erstellte Instanzen handelt.

SpecifyKind: Art des **DateTime**-Objekts festlegen. Der erste Parameter nimmt das **DateTime**-Objekt entgegen, wobei es sich um ein bestehendes oder auch um ein neu erstelltes **DateTime**-Objekt handeln kann. Der zweite Parameter erwartet ein Element der Enumeration **DateTimeKind**, also etwa **UTC**, **Local** oder **Unspecified**.

Today: Liefert ein **DateTime**-Objekt mit dem aktuellen Datum. Beispiel: **DateTime.Today.ToString()**

TryParse: Erwartet eine Zeichenkette mit einem Datum, welches in ein **DateTime**-Objekt konvertiert wird, das mit dem zweiten Parameter zurückgegeben wird. Die Funktion liefert den Wert **true** zurück, wenn die Zeichenkette geparst werden konnte, sonst false.

TryParseExact: Wie **TryParse**/**ParseExact**.

UTCNow: Liefert ein **DateTime**-Objekt, welches das aktuelle Datum und die Uhrzeit für die koordinierte Weltzeit enthält, also für die Zeitzone, in der sich Greenwich/London befindet.

Zur koordinierten Weltzeit

Die von der Funktion **UTCNow** gelieferte koordinierte Weltzeit liefert die aktuelle Uhrzeit plus Datum aus Greenwich (früher GMT/Greenwich Mean Time, jetzt UCT/Universal Coordinated Time). Wozu benötigt man diese Zeit, die sich zum Beispiel während der Sommerzeit um zwei Stunden von der Zeit in Deutschland unterscheidet? Wenn Sie eine Datenbank nutzen, die von Benutzern in verschiedenen Zeitzonen verwendet wird, dann sollten Sie Datumsangaben in der Datenbank in dieser Zeit speichern.

Alternativ müssen Sie die Zeitzone in einem separaten Feld speichern, damit Sie später beispielsweise nachvollziehen können, in welcher Reihenfolge Bestellungen eingegangen sind. Sonst würde eine Bestellung, die um 12:00 Uhr lokaler Zeit in Deutschland eingegangen ist gegenüber einer Bestellung, die um 9:00 Uhr lokaler Zeit an der amerikanischen Ostküste eingegangen ist, in der falschen Reihenfolge gespeichert werden – vorausgesetzt, die Zeitzone wurde nicht mitgespeichert. Wenn Sie die Zeiten in UTC-Zeit umwandeln, würde 12:00 Uhr deutscher Zeit 10:00 Uhr UTC entsprechen und 9:00 Uhr lokaler Zeit an der amerikanischen Ostküste wäre 13:00 Uhr UTC.

DateTime-Objekte per Konstruktor erstellen

Neben den Methoden wie **Now**, **Today** oder **UTCNow**, die automatisch das aktuelle Datum beziehungsweise die aktuelle Uhrzeit ermitteln, und Methoden wie **Parse** oder **TryParse**, denen Sie einen Datumsstring übergeben, der in ein **DateTime**-Objekt umgewandelt werden soll, gibt es noch verschiedene Konstruktor-Methoden der **DateTime**-Klasse. Das heißt, dass Sie die dem Datum und/oder der Uhrzeit zugrundeliegenden Informationen beim Initialisieren als Parameter übergeben.

Dazu haben wir ein kleines Beispiel erstellt, dass Sie auf der Registerseite **DateTime per Konstruktor** des Beispielprojekts finden. Hier kann der Benutzer Tag, Monat und Jahr eingeben und erhält das Datum in der kurzen Version. Dazu ermitteln wir den

int-Wert des jeweiligen Inhalts der Textfelder mit der **Parse**-Methode und übergeben Jahr, Monat und Tag dann dem Konstruktor von **DateTime**. Dieser landet dann wiederum im entsprechenden Textfeld:

```
private void btnDateTime_TagMonatJahr_Click(object sender, RoutedEventArgs e) {
    int tag = int.Parse(txtTag_TagMonatJahr.Text);
    int monat = int.Parse(txtMonat_TagMonatJahr.Text);
    int jahr = int.Parse(txtJahr_TagMonatJahr.Text);
    DateTime datAktuell = new DateTime(jahr, monat, tag);
    txtDateTime_TagMonatJahr.Text = datAktuell.ToShortDateString();
}
```

Das Gleiche erledigen wir direkt darunter mit der Uhrzeit. Dabei müssen wir auf jeden Fall auch ein gültiges Datum angeben, da es keine Überladung des Konstruktors gibt, der die Uhrzeit in Stunden, Minuten und Sekunden annimmt, aber kein Datum.

Hier verwenden wir **ToLongTimeString**, damit auch die Sekunden angezeigt werden:

```
private void btnDateTime_StundeMinuteSekunde_Click(object sender, RoutedEventArgs e) {
    int stunde = int.Parse(txtStunde_StundeMinuteSekunde.Text);
    int minute = int.Parse(txtMinute_StundeMinuteSekunde.Text);
    int sekunde = int.Parse(txtSekunde_StundeMinuteSekunde.Text);
    DateTime datUhrzeit = new DateTime(1, 1, 1, stunde, minute, sekunde);
    txtDateTime_StundeMinuteSekunde.Text = datUhrzeit.ToLongTimeString();
}
```

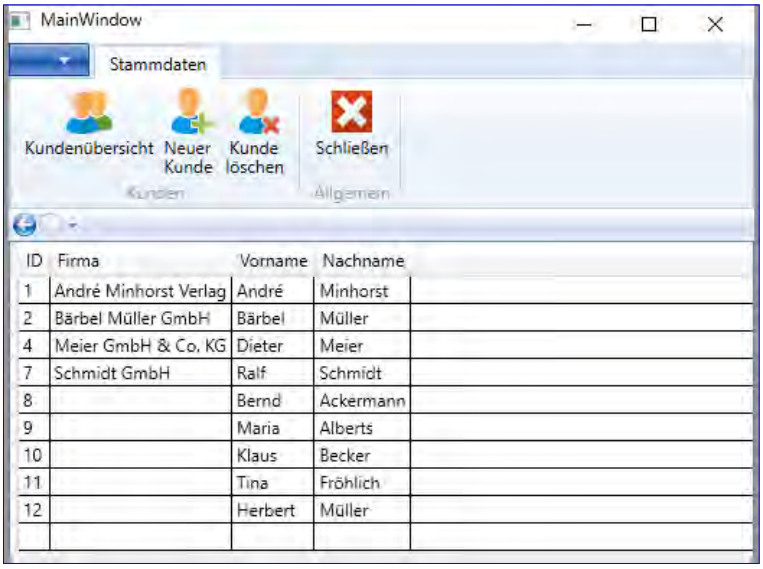
Die verschiedenen Überladungen des Konstruktors nehmen die folgenden Parameter entgegen (die Varianten mit Umwandlungen in andere Kalender als den gregorianischen lassen wir weg):

- Kein Parameter: Liefert das Datum 01.01.0001 00:00:00.
- Anzahl der Ticks (long): Speichert direkt den **long**-Wert in **DateTime**.
- Ticks und **DateTimeKind**: Speichert die Ticks unter Anwendung eines weiteren Parameters, der angibt, ob die Zeit als lokale Zeit (**DateTimeKind.Local**, **2**) oder als UTC-Zeit (**DateTimeKind.UTC**, **1**) gespeichert werden soll.
- Jahre, Monate, Tage
- Jahre, Monate, Tage, Stunden, Minuten, Sekunden
- Jahre, Monate, Tage, Stunden, Minuten, Sekunden und **DateTimeKind**
- Jahre, Monate, Tage, Stunden, Minuten, Sekunden, Millisekunden

DataGrid im Detail

Bisher haben wir das DataGrid immer in einer Minimalausführung verwendet – also mit einfachen Text-Steuerelementen. Das sieht nicht nur langweilig aus, sondern es nutzt auch die Fähigkeiten des DataGrid-Steuerelements nicht annähernd aus. Also schauen wir uns in diesem Artikel an, welche Möglichkeiten der Formatierung des DataGrid-Steuerelements wie bisher in unserer Lösung »Bestellverwaltung« eingesetzt haben.

Um genau zu sein, haben wir das DataGrid in den bisherigen Beispielen so genutzt wie ein einfaches **ListView**-Steuerelement – wir haben ihm eine Datenherkunft zugewiesen und die enthaltenen Daten einfach in den einzelnen Spalten des Steuerelements angezeigt (siehe Bild 1).



ID	Firma	Vorname	Nachname
1	André Minhorst Verlag	André	Minhorst
2	Bärbel Müller GmbH	Bärbel	Müller
4	Meier GmbH & Co. KG	Dieter	Meier
7	Schmidt GmbH	Ralf	Schmidt
8		Bernd	Ackermann
9		Maria	Alberts
10		Klaus	Becker
11		Tina	Fröhlich
12		Herbert	Müller

Bild 1: Einfaches DataGrid-Steuerelement

Hinweis

Die folgenden Beispiele basieren auf der Verwendung eines Entity Data Models der Datenbank **Bestellverwaltung** als Datenquelle.

Datenquellen

Normalerweise zeigt das DataGrid automatisch alle Spalten und alle Zeilen der Datenquelle an. Eine individuelle Anzeige erreichen Sie, indem Sie zunächst die automatische Anzeige aller Spalten deaktivieren. Dazu stellen Sie das Attribut **AutoGenerateColumns** auf **False** ein. Sie müssen dann individuelle Spalten angeben – siehe weiter unten.

Rasterlinien

Die Rasterlinien werden standardmäßig als dünne, schwarze Linien angezeigt. Die generelle Sichtbarkeit können Sie mit dem Attribut **GridLinieVisibility** beeinflussen, das die Werte **All**, **Horizontal**, **None** und **Vertical** entgegennimmt. Die Farbe der Rasterlinien lässt sich mit den Eigenschaften **HorizontalGridLinesBrush** und **VerticalGridLinesBrush** individuell einstellen:

```
<DataGrid ItemsSource="{Binding Kunden}" HorizontalGridLinesBrush="LightGray" VerticalGridLinesBrush="LightGray">
```

Standardfunktionen

Das DataGrid bietet schon in der Grundausstattung einige Funktionen, die etwa in der Datenblattansicht von Access enthalten sind – aber anders genutzt werden können. Diese Möglichkeiten können Sie nach Bedarf einschränken, indem Sie das entsprechende in Klammern angegebene Attribut auf den Wert **False** einstellen:

- Sortieren der Zeilen nach dem Inhalt einer Spalte. Dazu klicken Sie mit der Maus auf den Kopf der Spalte, nach welcher die Daten sortiert werden sollen. Der erste Klick sortiert die Daten in aufsteigender, der nächste in absteigender Reihenfolge (**CanUserSortColumns**).

- Sortieren der Spalten: Sie können die Reihenfolge der Spalten ganz einfach per Drag and Drop einstellen (**CanUserReorderColumns**).

- Ändern der Spaltenbreiten: Per Ziehen des Trennstrichs rechts von einem Spaltenkopf ändern Sie die Breite dieser Spalte (**CanUserResizeColumns**).

- Außerdem können Sie auch die Zeilenhöhe anpassen, indem Sie auf den Strich unter dem grauen Bereich links von der ersten Spalte nach oben oder unten ziehen (**CanUserResizeRows**).

Textspalten

Wenn Sie **AutoGenerateColumns** auf **False** eingestellt haben, müssen Sie selbst angeben, welche Steuerelementtypen für welches Feld der Datenherkunft im DataGrid angezeigt werden sollen. Wenn die Spalten automatisch generiert werden, kommt das zum Datentyp der anzuzeigenden Eigenschaft zum Einsatz. Wenn Sie die folgenden vier Felder der Datenherkunft anzeigen wollen, würden Sie dazu vier **DataGridTextColumn**-Elemente verwenden:

```
<DataGrid ItemsSource="{Binding Kunden}" AutoGenerateColumns="False" HorizontalGridLinesBrush="LightGray"
        VerticalGridLinesBrush="LightGray">
    <DataGrid.Columns>
        <DataGridTextColumn Binding="{Binding ID}" Header="ID"></DataGridTextColumn>
        <DataGridTextColumn Binding="{Binding Firma}" Header="Firma"></DataGridTextColumn>
        <DataGridTextColumn Binding="{Binding Vorname}" Header="Vorname"></DataGridTextColumn>
        <DataGridTextColumn Binding="{Binding Nachname}" Header="Nachname"></DataGridTextColumn>
    </DataGrid.Columns>
</DataGrid>
```

Dabei geben Sie mit **Binding="{Binding <Feld>}"** das zu bindende Feld der Datenquelle an und mit **Header** die Spaltenüberschrift.

Andere Spaltentypen

Es gibt noch eine ganze Reihe weiterer Spaltentypen, mit denen Sie Felder mit anderen Datentypen besser als mit einem Textfeld anzeigen können:

- **DataGridCheckBoxColumn**: Kontrolllkästchen, zeigt **True** (beziehungsweise **1**) als markiert und **False** (**0**) als nicht markiert an – siehe weiter unten.

- **DataGridComBoxColumn**: Kombinationsfeld. Dieses Element hat, wie weiter unten sehen werden, gerade bei der Datenbindung eine Besonderheit.

- **DataGridHyperlinkColumn**: Zeigt den enthaltenen Text als Link an.

- **DataGridTemplateColumn**: Spaltentyp zum Definieren eigener Vorlagen, auch mit anderen Steuerelementen als **TextBox**, **CheckBox**, **ComboBox** und **Hyperlink**

DataGrid mit CheckBox

Auf der Seite **Produktuebersicht.xaml** der Beispielanwendung **Bestellverwaltung** haben wir ein Kontrollkästchen zum DataGrid hinzugefügt. Die Definition sieht wie folgt aus, die Ansicht zur Laufzeit finden Sie in Bild 2:

```
<DataGrid x:Name="dgProdukte" Grid.Column="0" Grid.Row="0" Grid.ColumnSpan="2" ItemsSource="{Binding Produkte}"
AutoGenerateColumns="false" CanUserAddRows="False">
    <DataGrid.Columns>
        <DataGridTextColumn Binding="{Binding Path=ID}" Header="ID"  />
        <DataGridTextColumn Binding="{Binding Path=Bezeichnung}" Header="Produkt" />
        <DataGridCheckBoxColumn Binding="{Binding Path=Lieferbar}" Header="Lieferbar" />
    </DataGrid.Columns>
    ...
</DataGrid>
```

DataGrid mit ComboBox

Während die Definition von Textfeldern und Kontrollkästchen im DataGrid relativ einfach ist, wird es beim Kombinationsfeld etwas komplizierter: Hier können Sie nämlich nicht einfach die Eigenschaften wie **ItemsSource**, **SelectedItem**, **DisplayMemberPath** oder **SelectedValuePath** verwenden. Schauen wir uns die Grundlagen für die Anzeige etwa der Kategorien in der Seite **Produktuebersicht.xaml** an. Hier benötigen wir in der Code behind-Klasse eine Liste mit den Kategorien:

```
private List<Kategorie> kategorien;
public List<Kategorie> Kategorien {
    get { return kategorien; }
    set { kategorien = value; }
}
```

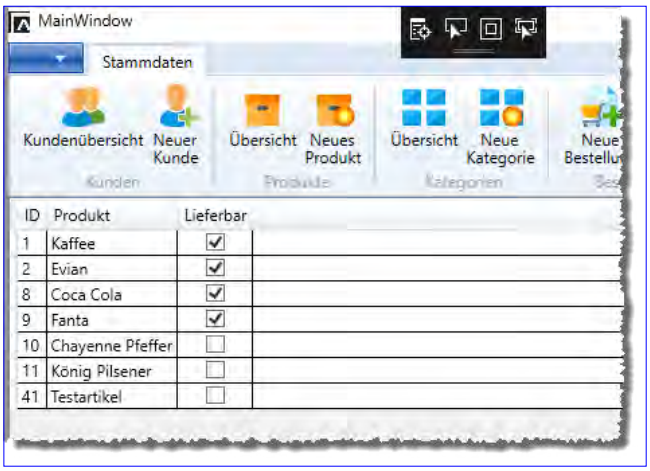


Bild 2: DataGrid mit Kontrollkästchen

Diese Liste müssen Sie natürlich noch initialisieren, was in der Konstruktor-Methode geschieht:

```
public Produktuebersicht(long kundeID = 0) {
    ...
    kategorien = new List<Kategorie>(dbContext.Kategorien);
}
```

Nun kommt der interessante Teil, nämlich die Definition des Kombinationsfeldes in der **xaml**-Datei:

```
<DataGridComboBoxColumn Header="Kategorie" SelectedValueBinding="{Binding KategorieID}" DisplayMemberPath="Bezeichnung"
        SelectedValuePath="ID">
    <DataGridComboBoxColumn.ElementStyle>
```

Webservice testen am Beispiel von DHL

Wer einen Webshop betreibt, kennt sich zumindest mit den Webseiten des einen oder anderen Paketdienstes aus. Dabei gibt es beim Erstellen von Versandetiketten für Pakete verschiedene Stufen der Automatisierung. Wir wollen zur Stufe der höchsten Automatisierung gelangen und unsere Daten für den Versand direkt an den Webservice eines Anbieters, in diesem Fall DHL, weiterleiten und die erstellten Versandetiketten in druckbarer Form herunterladen. Dazu erfahren Sie in diesem Artikel, wie Sie einen Webservice mithilfe des Tools SoapUI testen.

Von der Handarbeit zur Vollautomatisierung

Wer es nicht so mit dem Internet hat, holt sich ein paar Paketscheine von der Post, füllt diese händisch mit den Adressen der Empfänger aus und sendet diese dann ab. Bei kleinen Mengen mag das ein sinnvoller Ansatz sein. Wer hingegen gleich ein paar Lieferungen pro Tag verschickt, gibt die Daten vielleicht direkt in die Eingabemaske der Webseite **dhl.de** ein. Hier können Sie alle relevanten Daten angeben – von der Paketgröße über das Gewicht bis hin zu den Absender- und Empfängerdaten. Wer regelmäßig Pakte versendet, möchte aber vielleicht noch etwas Geld sparen und nutzt den Geschäftskundentarif statt den für Privatkunden. Dann ist **www.dhl-geschaeftskundenportal.de** die richtige Adresse. Hier benötigen Sie allerdings ein Kundenkonto – auf die Schnelle ein Paket verschicken und per Paypal bezahlen können Sie hier nicht. Als registrierter Kunde loggen Sie sich dann auf der Seite ein und können über den Menüpunkt **VersendenlVersenden (Intraship)** zur Versandschnittstelle wechseln (siehe Bild 1).

Bild 1: Aufruf des Dienstes **Intraship**

Hinweis: Zum Testen der in späteren Beispielen vorgestellten Zugriffe auf den Webservice von DHL benötigen Sie ein Entwickler-Konto bei DHL. Wie Sie dieses anlegen, erfahren Sie weiter unten.

Bild 2: Versandschnittstelle

Hier haben Sie dann einige weitere Möglichkeiten. Dazu gehört das Anlegen eines neuen Auftrags, wobei Sie allerdings nur das Gewicht und die Empfängeradresse eintragen müssen – die Absenderadresse für Ihr Kundenkonto wird in den Einstellungen gespeichert

Bild 3: Auswahl einer Empfängerliste

und automatisch hinzugefügt. Außerdem können Sie hier einen Import von Aufträgen durchführen, wozu sie auf den entsprechenden Link klicken (siehe Bild 2).

Damit landen Sie auf einer Seite, auf der Sie die Datei mit den dafür vorbereiteten auswählen und die enthaltenen Daten einlesen können. Nach dem Import zeigt die Seite die eingelesenen Sendungen in einer Liste an und Sie können diese markierten und ausdrucken oder auch stornieren. Nach dem Ausdrucken öffnet sich ein weiteres Fenster mit einer PDF-Datei, welche die Versandetiketten enthält. Diese drucken Sie nun noch auf entsprechendem Papier aus. Dieses stellt DHL seinen Geschäftskunden kostenlos zur Verfügung (siehe Bild 3).

Nun wollen wir dies alles komplett automatisieren. Das heißt, dass wir gar keinen Internetbrowser mehr öffnen wollen, sondern aus unserer Anwendung heraus einen Webservice aufrufen wollen, dem wir die Absender- und Empfängerdaten und die Angaben zur Sendung übermitteln wollen und der uns die Etiketten dann in der Antwort zur Verfügung stellen soll.

DHL für Entwickler

Damit landen wir dann auf der Seite **entwickler.dhl.de**. Hier können Sie nun unkompliziert ein Entwicklerkonto erstellen – die einzelnen Schritte wollen wir hier nicht darstellen. Nach dem Anlegen des Entwicklerkontos und dem

Bild 4: Einige der verfügbaren APIs

Anmelden im Portal finden Sie eine Auswahl der verfügbaren APIs (siehe Bild 4). Uns interessiert im Moment vor allem die API für den Geschäftskundenversand.

Webservice ansehen

Wenn Sie sich in den Informationen zur API für Geschäftskunden umsehen, finden Sie zunächst heraus, wo Sie die WSDL-Datei des Webservices finden, nämlich unter folgendem Link:

```
https://cig.dhl.de/cig-wsdls/com/dpdhl/wsdl/geschaeftskunden-
versand-api/2.2/geschaeftskundenversand-api-2.2.wsdl
```

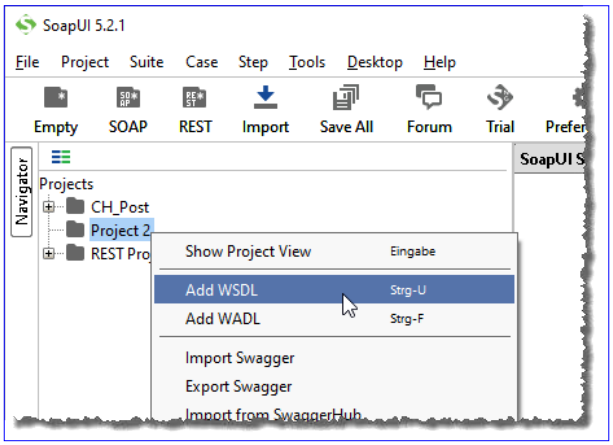


Bild 5: Hinzufügen einer WSDL

Was Sie damit alles anstellen können, schauen wir uns mit einem speziellen Tool namens SoapUI an, mit dem Sie den Webservice auch testen können, ohne auch nur eine Zeile Code zu programmieren. Dieses laden Sie von der Seite **https://www.soapui.org/** herunter und installieren es. Danach starten Sie SoapUI und klicken auf den Menüeintrag **File|Create Empty Project**.

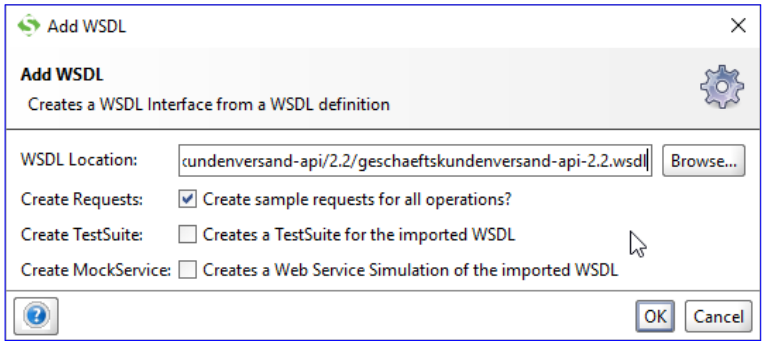


Bild 6: WSDL eingeben

Klicken Sie dann mit der rechten Maustaste auf den neuen Eintrag und wählen Sie den Kontextmenü-Eintrag **Add WSDL** aus (siehe Bild 5).

Es erscheint ein kleiner Dialog, in dem Sie unter **WSDL Location** den Link mit der WSDL-Datei eintragen (siehe Bild 6). Behalten Sie die Einstellung **Create sample requests for all Operations?** bei und klicken Sie auf **OK**.

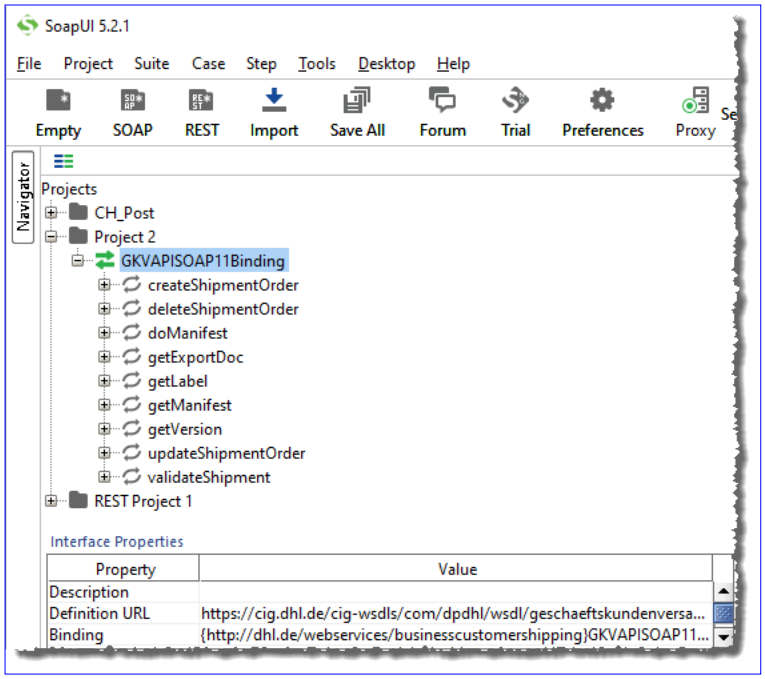


Bild 7: Funktionen des Webservices

Anschließend zeigt **SoapUI** im Bereich **Projects** alle Funktionen des Webservices mit der soeben analysierten WSDL-Datei an (siehe Bild 7).

Nun wollen wir einen Request zum Testen der Funktion **getLabel** des Webservice erstellen – **getLabel** hört sich so an, also ob es unsere Aufgabe, ein Label auf Basis der Versanddaten zu erstellen, erfüllen könnte. Dazu klicken Sie wieder mit der rechten Maustaste auf **getLabel**

```
<soapenv:Envelope xmlns:soapenv="...">
   <soapenv:Header>
      <cis:Authentification>
         <cis:user>?</cis:user>
         <cis:signature>?</cis:signature>
      </cis:Authentification>
   </soapenv:Header>
   <soapenv:Body>
      <bus:GetLabelRequest>
         <bus:Version>
            <majorRelease>?</majorRelease>
            <minorRelease>?</minorRelease>
            <!--Optional:-->
            <build>?</build>
         </bus:Version>
         <!--1 to 30 repetitions:-->
         <cis:shipmentNumber>?</cis:shipmentNumber>
         <!--Optional:-->
         <labelResponseType>?</labelResponseType>
      </bus:GetLabelRequest>
   </soapenv:Body>
</soapenv:Envelope>
```

Listing 1: Request **getLabel** mit Platzhaltern

und wählen den Eintrag **New request** aus dem Kontextmenü aus. Geben Sie nun als Namen **getLabelRequest** ein und schauen Sie sich den Inhalt des nun geöffneten Fensters an.

Hier finden wir ein Element vor, dass die für die Authentifizierung notwendigen Daten entgegennimmt und einen Body, in den Sie eine Version eingeben und bis zu 30 **shipmentNumbers** eingeben können. Das sieht doch eher danach aus, als ob man hier bereits erstellte Label anfragen kann statt neue anzulegen. Also müssen wir wohl doch noch einen Blick auf die übrigen Funktionen des Webservice werfen.

Request testen

Schnell zeigt sich, dass **CreateShipment-Order** die passende API-Funktion ist. Also erstellen wir einen neuen Request auf Basis diese Funktion. Das schöne am Tool SoapUI

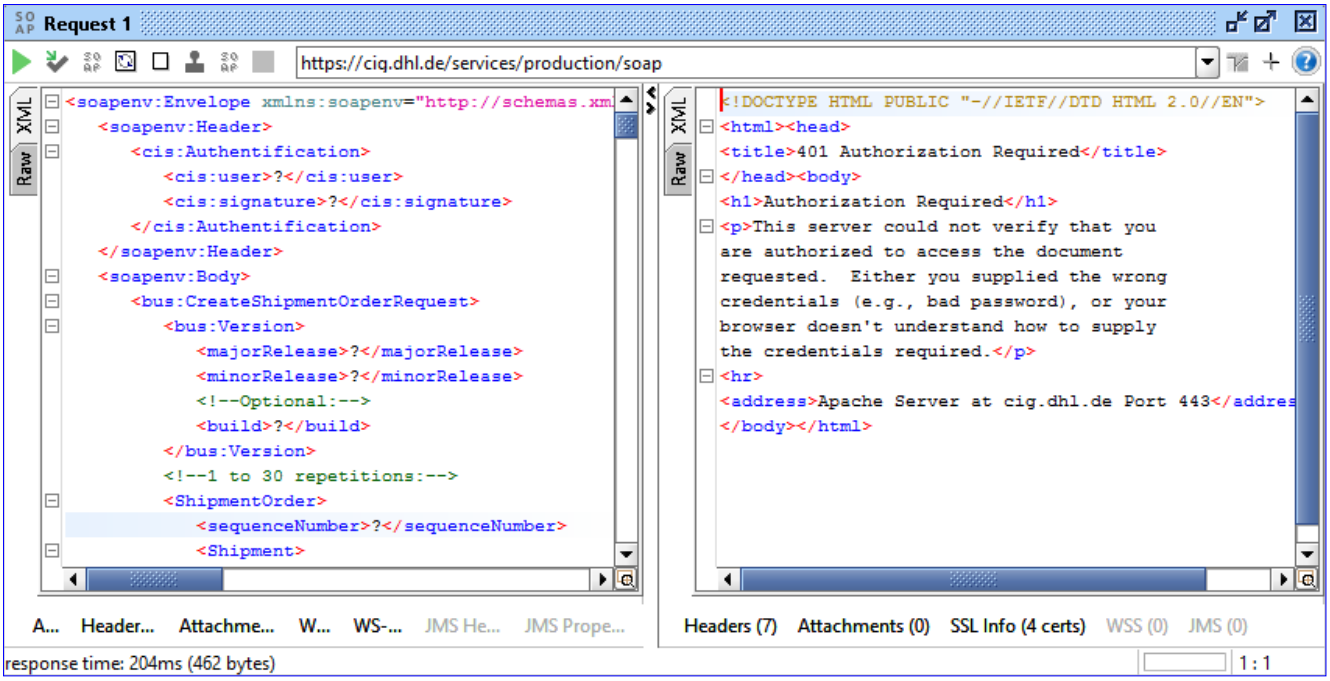


Bild 8: Absenden eines ersten Requests

ist: Sie können bereits jetzt einfach einmal einen Request absenden und erhalten die Antwort direkt im gleichen Fenster. In diesem Fall weist uns die Antwort darauf hin, dass wir keine Authentifizierungsdaten eingegeben haben (siehe Bild 8).

Also kümmern wir uns erst einmal um diese Daten. Diese finden Sie in der Onlinedokumentation. In diesem Fall heißt der Benutzer **2222222222_01** und das Kennwort **pass**. Außerdem müssen wir den Endpunkt für den Webservice ändern, denn nach dem Einlesen der WSDL-Datei lautet dieser auf dem Endpunkt für den Produktivbetrieb. Zum Testen gibt es natürlich einen eigenen Endpunkt, der **https://cig.dhl.de/services/sandbox/soap** lautet und den Sie in SoapUI für die Eigenschaft **Endpoint** in den **Request Properties** einstellen. Allerdings ist der Sandbox-Endpunkt dort noch nicht vorhanden, weshalb Sie diesen noch hinzufügen müssen.

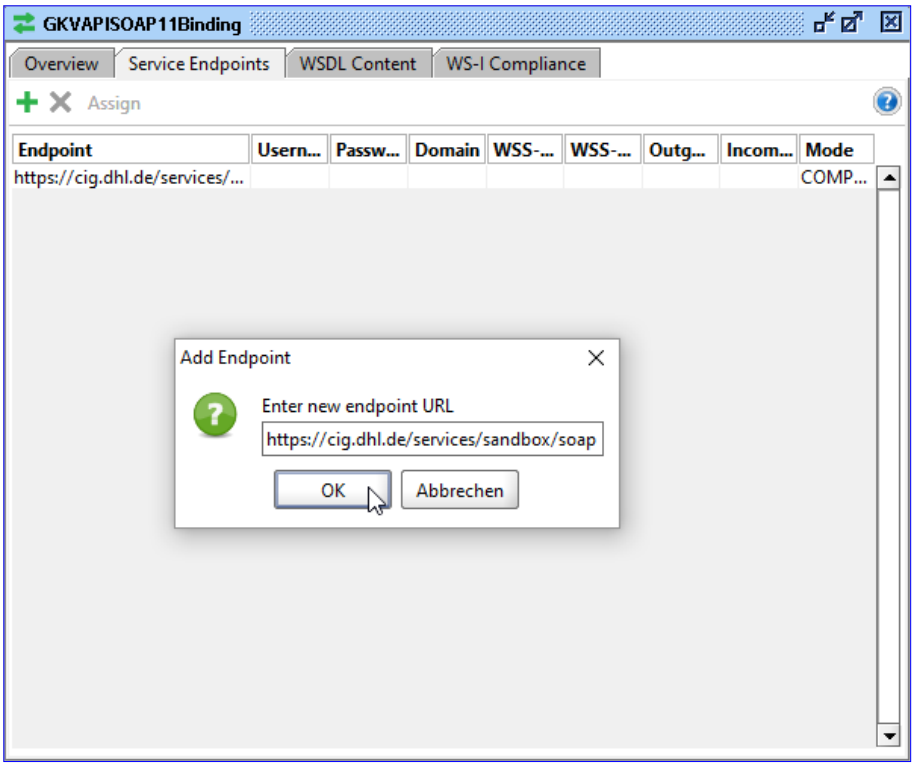


Bild 9: Neuen Endpoint hinzufügen

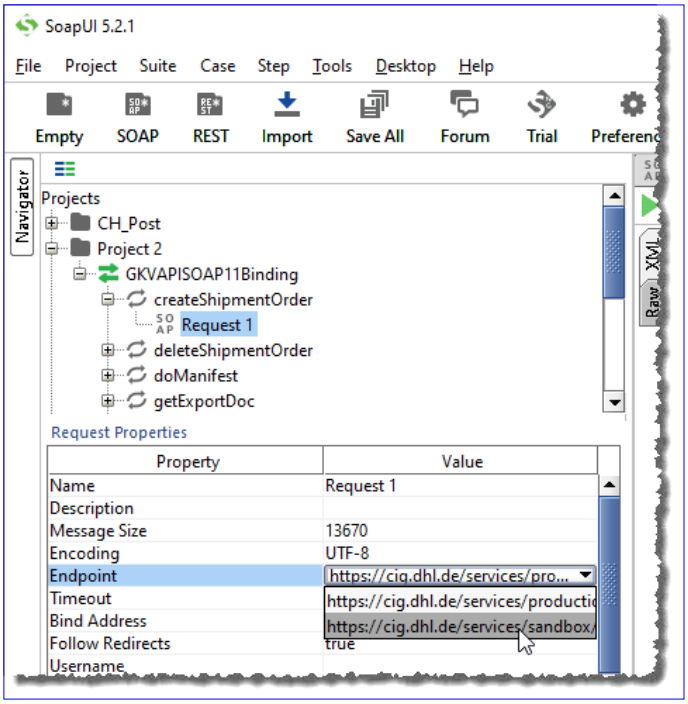


Bild 10: Neuen Endpoint auswählen

Endpunkt zum Testen anlegen

Dazu klicken Sie im Bereich **Navigator** doppelt auf das Element unterhalb des Projekt-Elements, dass in unserem Beipsiel **GKVAPISOAP11Binding** genannt wurde. Es erscheint ein gleichnamiger Dialog, in dem Sie auf die Registerseite **Service Endpoints** wechseln.

Hier klicken Sie auf das Plus-Symbol, um einen neuen Endpoint anzulegen und tragen dort den Wert **https://cig.dhl.de/services/sandbox/soap** ein (siehe Bild 9).

Neuen Endpunkt auswählen

Nach dem Schließen des Dialogs können Sie in den Eigenschaften des Requests auf Basis von **createShipmentOrder** den neuen Endpunkt auswählen (siehe Bild 10).

Webservice mit C# am Beispiel von DHL-Etiketten

Mit Webservices lässt sich der Abruf von Daten bestimmter Anbieter aus dem Internet stark vereinfachen. Während Sie etwa zum Herunterladen eines Versandetiketts einige Minuten investieren müssen, um die Adressen und weitere Sendungsdaten aus einer Kundendatenbank in ein Onlineformular zu übertragen, könnten Sie das auch per Mausklick erledigen. Voraussetzung sind einige Zeilen Code und die Registrierung beim entsprechenden Webservice. In diesem Fall schauen wir uns das am Beispiel von Versandetiketten des Paketdienstleisters DHL an.

Im Artikel Webservice testen am Beispiel von DHL haben wir uns bereits angesehen, wie Sie die für einen Webservice wichtige WSDL-Datei ermitteln und damit über ein Tool wie SoapUI die XML-Vorlage für einen Request ermitteln, füllen und testen. Dort haben wir außerdem ein Entwickler-Konto angelegt, mit dem wir auf den Webservice von DHL zugreifen können. Schließlich haben wir einen funktionierenden Aufruf an die Sandbox-Version des Webservice, also eine zu Testzwecken bestimmte Umgebung, abgesetzt und mit dem zurückgelieferten Link ein PDF-Dokument mit dem Versandetikett erhalten (siehe Bild 1).

Bild 1: Aufruf des Dienstes **Intraship**

Nun wollen wir uns die Techniken ansehen, die dafür nötig sind, einen Request an einen Webservice abzusenden und die Antwort auszuwerten. Hier interessiert uns im vorliegenden Beispiel natürlich vor allem, wie wir den Link zum PDF-Dokument mit dem Versandetikett verarbeiten und das PDF-Dokument auf der Festplatte speichern oder ausdrucken.

Anfrage zusammenstellen und abschicken

Wir wollen zunächst den XML-Request, den wir im Artikel Webservice testen am Beispiel von DHL ermittelt haben, an den Webservice schicken und den Request entgegennehmen. Dabei müssen wir natürlich auch noch die Authentifizierung berücksichtigen. Um die Funktion zum Ermitteln des Response-Dokuments aufzurufen, verwenden wir ein Fenster in einer WPF-Anwendung, das wir mit einer entsprechenden Schaltfläche ausstatten.

Diese gibt dann einfach den Code in einem Meldungsfenster aus (siehe Listing 1).

```
private void btnEinfacherAufruf_Click(object sender, RoutedEventArgs e) {
    MessageBox.Show(GetResponse());
}
```

Listing 1: Aufruf der Funktion zum Senden eines Requests an den Webservice

Um die verschiedenen nachfolgend verwendeten Klassen zu nutzen, die sich auf den Umgang mit dem Webservice, XML-Dokumenten und das Lesen und Schreiben von Dokumenten beschäftigen, benötigen wir noch drei Namespaces, die wir mit dem **using**-Schlüsselwort einbinden:

```
using System.Net;
using System.Xml;
using System.IO;
```

Die Funktion **GetResponse**, die den Ablauf steuert, finden sieht wie folgt aus:

```
public string GetResponse() {
    HttpWebRequest request = CreateRequest();
    XmlDocument requestXML = GetRequestXML();
    using (Stream stream = request.GetRequestStream()) {
        requestXML.Save(stream);
    }
    using (WebResponse response = request.GetResponse()) {
        using (StreamReader rd = new StreamReader(response.GetResponseStream())) {
            var result = rd.ReadToEnd();
            return result;
        }
    }
}
```

Hier ermitteln wir mit der Funktion **CreateRequest** ein neues Objekt des Typs **HttpWebRequest** und speichern es in der Variablen **request**. Dabei werden Informationen wie die Header für den Aufruf, die Authentifikation und weitere Informationen in einem Objekt zusammenstellt – mehr dazu weiter unten. Danach füllen wir ein Objekt des Typs **XmlDocument** namens **requestXML**, und zwar ebenfalls mithilfe einer Funktion. Diese heißt **GetRequestXML** und stellt einfach nur das XML-Dokument zusammen, das als Inhalt des Requests verwendet werden soll. Wir haben also nun ein Objekt des Typs **HttpWebRequest**, welches die Daten für die Steuerung des Requests enthält, und ein XML-Dokument mit dem zu übergebenen Request.

Nun müssen wir beide noch verheiraten. Dies geschieht im ersten **using**-Konstrukt der Funktion **GetResponse**. Das in Klammern hinter der **using**-Anweisung erzeugte Objekt wird nach dem Durchlaufen des **using**-Abschnitts wieder gelöscht. In diesem Fall handelt es sich um ein **Stream**-Objekt. Diesem weisen wir das mit der Funktion **GetRequestStream** ermittelte **Stream**-Objekt unseres **HttpWebRequest** namens **request** zu. Wir haben also mit **stream** ein **Stream**-Objekt referenziert, dass wir nun füllen können und so den übergebenen Inhalt als Request-Inhalt an das **HttpWebRequest**-Objekt übergeben. Dies erledigen wir mit der einzigen Anweisung des **using**-Konstrukts, mit der wir das in **requestXML** gespeicherte XML-Objekt in den Stream und somit in das Objekt **request** speichern.

Das zweite **using**-Konstrukt ruft dann bereits die Methode **GetResponse** von **request** auf und speichert das Ergebnis in der Variablen **response** mit dem Datentyp **WebResponse**. resonse liefert mit **GetResponseStream** wiederum einen Stream, den wir

diesmal allerdings nicht in einem **Stream**-Objekt speichern, sondern einem **StreamReader** als Konstruktorparameter übergeben. Wir erstellen also ein neues **StreamReader**-Objekt und weisen diesem den Stream zu, den wir als Antwort des Webservice erhalten haben. Diesen in **rd** gespeicherten Streamreader kapseln wir wiederum in einem **using**-Konstrukt, da wir ihn nur für die folgenden beiden Anweisungen benötigen. Die erste liest mit **ReadToEnd** den kompletten Inhalt des in **rd** gespeicherten Streams und fügt ihn der Variablen **result** hinzu. **result**, dass zu diesem Zeitpunkt das XML-Dokument mit der Antwort des Webservices enthält, wird dann schließlich als Funktionswert zurückgegeben. Die aufrufende Methode gibt den Inhalt dann schließlich wie in Bild 2 als Meldung aus.

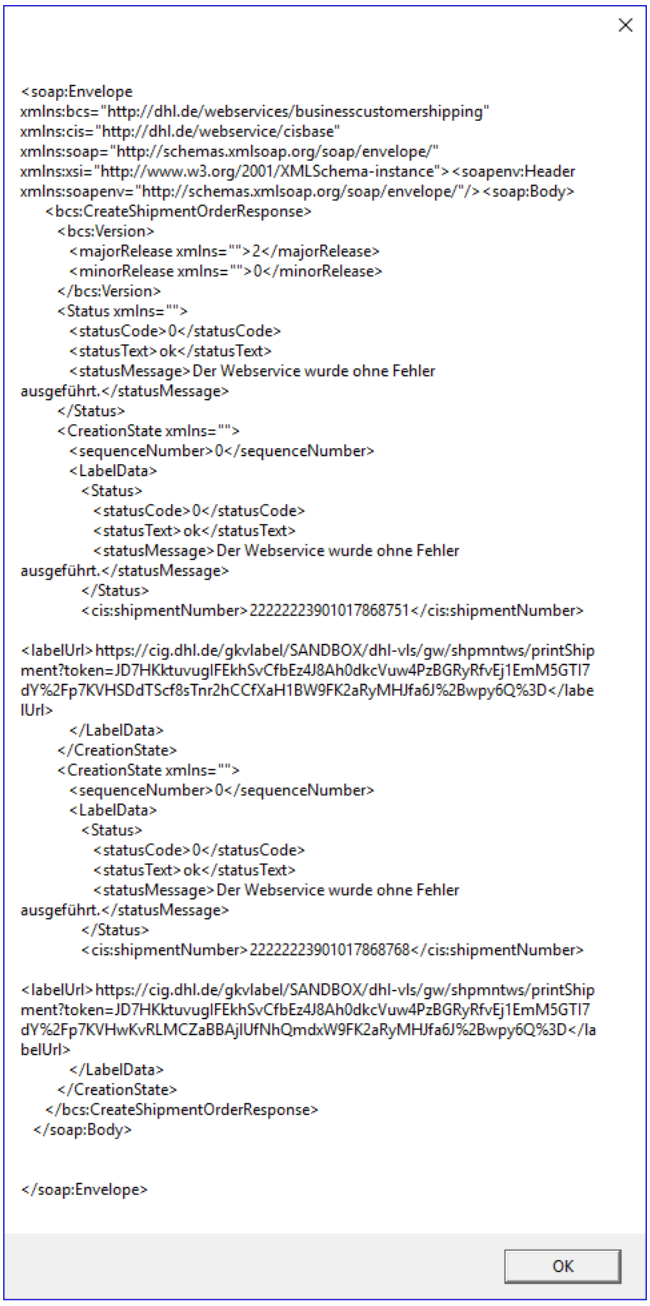


Bild 2: Ergebnis des Requests

Request zusammenstellen

Die Funktion **CreateRequest** stellt den Request zusammen und gibt diesen als **HttpWebRequest**-Objekt an die aufrufende Instanz zurück. Dabei legt diese zunächst ein **Uri**-Objekt an und hinterlegt die Uri, die als Endpunkt für den Webservice dient (hier der Request für die Sandbox: **https://cig.dhl.de/services/sandbox/soap**). Danach erstellt sie das **HttpWebRequest**-Objekt auf Basis der Methode **Create** der Klasse **WebRequest**. Dieser übergibt sie das **Uri**-Objekt mit dem Endpunkt als Parameter. Das Ergebnis wird schließlich in den Typ **HttpWebRequest** umgewandelt. Danach stellt die Funktion die Zeichenkette für die Authentifizierung zusammen. Der Benutzername und das Kennwort werden zunächst in die Variablen **username** und **password** geschrieben (später fragen Sie die Daten entweder ab oder speichern direkt die kodierte Form), durch einen Doppelpunkt voneinander getrennt in einer neuen Zeichenkette aufgenommen und in einen Base64-String umgewandelt. Das Ergebnis landet in der Variablen **auth**. Danach fügt die **Add**-Methode der **Headers**-Auflistung des in **request** gespeicherten **WebHttpRequest**-Objekts einen neuen Header mit dem Namen **Authorization** und dem aus der Zeichenkette **Basic**, einem Leerzeichen und der kodierten Version aus Benutzername und Kennwort als Wert hinzu. Danach legt sie noch die Eigenschaften **ContentType** (**text/xml;charset="utf-8"**), **Accept** (**text/xml**) und **Method** (**POST**) von **request** fest. Danach gibt die Methode den Inhalt der Objektvariablen request an die aufrufende Methode zurück:

```
public HttpWebRequest CreateRequest() {
    Uri uri = new Uri("https://cig.dhl.de/services/sandbox/soap");
```

```
    HttpWebRequest request = (HttpWebRequest)WebRequest.Create(@uri);
    String username = "andreminhorst";
    String password = "*********";
    String auth = System.Convert.ToBase64String(System.Text.Encoding.UTF8.GetBytes(username + ":" + password));
    request.Headers.Add("Authorization", "Basic " + auth);
    request.ContentType = "text/xml;charset=\"utf-8\"";
    request.Accept = "text/xml";
    request.Method = "POST";
    return request;
}
```

XML-Dokument zusammenstellen

Damit kommen wir zur letzten Funktionsmethode, die wir für den ersten Aufruf benötigen. Diese heißt **GetRequestXML** und erstellt zunächst ein neues Objekt des Typs **XmlDocument**. Die Variable für dieses Objekt heißt **requestXML**.

Es soll nun mit dem als Zeichenkette vorhandenen XML-Dokument gefüllt werden, welches wir im Artikel **Webservice testen am Beispiel von DHL** zusammengestellt und ausprobiert haben. Diese Zeichenkette laden wir mit der Methode **LoadXml** in das Objekt **requestXML** und geben dieses als Rückgabewert an die aufrufende Methode zurück:

```
public XmlDocument GetRequestXML() {
    XmlDocument requestXML = new XmlDocument();
    requestXML.LoadXml(@"<soapenv:Envelope ...>
        <soapenv:Header>
                 <cis:Authentification>
                 <cis:user>2222222222_01</cis:user>
                <cis:signature>pass</cis:signature>
            </cis:Authentification>
        </soapenv:Header>
        ---
    </soapenv:Envelope>");
    return requestXML;
}
```

Damit haben wir mit nicht allzu vielen Zeilen einen funktionierenden Aufruf eines Webservice erstellt, der in seinem Response zum Beispiel den Link zu einer PDF-Datei mit dem gewünschten Versandetikett zurückliefert.

Nächste Schritte

Was wollen wir nun als nächstes erledigen? Folgende Aufgaben stehen noch an:

- Anpassen der Funktion **GetRequestXML**, damit wir das damit übergebene XML-Dokument dynamisch zusammenstellen können – beispielsweise auf Basis der Daten, die der Benutzer in ein Fenster eingibt oder der Daten einer Kunden-Tabelle

- Auslesen der Response-Datei, um den Link zum Versandetikett zu ermitteln
- Herunterladen des Dokuments, dass sich hinter diesem Link verbirgt
- Ausdrucken des Versandetiketts

Wir stellen das Anpassen der Funktion **GetRequestXML** nach hinten, weil wir zunächst die Funktionen zum Ermitteln, Herunterladen und Drucken des Versandetiketts programmieren wollen.

URL des Versandetiketts ermitteln

Nun wollen wir das XML-Dokument, dass uns als Response geliefert wurde, auseinandernehmen und die enthaltenen URLs zu den Versandetiketten ermitteln. Dazu wollen wir aus dem nachfolgend ausschnittweise dargestellten Response-XML-Dokument zunächst prüfen, ob der allgemeine **statusCode** den Wert **0** hat. Falls ja, durchlaufen wir die einzelnen **CreationState**-Elemente per Code und schauen uns dort den Wert des Label-spezifischen Elements **statusCode** an. Hat auch dieses den Wert **0**, wollen wir uns den Inhalt des Elements **sequenceNumber**, des Elements **cis:shipmentNumber** und des Elements **labelUrl** merken. Wie das genau geht, erfahren Sie gleich im Anschluss:

```
<soap:Envelope ...>
   <soapenv:Header xmlns:soapenv="http://schemas.xmlsoap.org/soap/envelope/"/>
   <soap:Body>
      <bcs:CreateShipmentOrderResponse>
         <bcs:Version>...</bcs:Version>
         <Status>...
            <statusCode>0</statusCode>
            ...
         </Status>
         <CreationState>
            <sequenceNumber>0</sequenceNumber>
            <LabelData>
               <Status>
                  <statusCode>0</statusCode>
                  ...
               </Status>
               <cis:shipmentNumber>22222223901017866665</cis:shipmentNumber>
               <labelUrl>https://cig.dhl.de/gkvlabel/SANDBOX/dhl-vls/gw/shpmntws/printShipment?token=JD7H...</labelUrl>
            </LabelData>
         </CreationState>
         <CreationState>
               ...
               <labelUrl>https://cig.dhl.de/gkvlabel/SANDBOX/dhl-vls/gw/shpmntws/printShipment?token=JD7HKk...</labelUrl>
               ...
```

PDFs zusammenführen mit iTextSharp

Im Rahmen unserer Lösung zum Erstellen von DHL-Versandetiketten benötigen wir eine Lösung, mit der wir mehrere PDF-Dokumente zu einem zusammenführen können, um dieses dann in einem Rutsch auszudrucken. Mit Bordmitteln kommen wir nicht weiter, aber es gibt ja eine Reihe von Bibliotheken, die man sich etwa als NuGet-Paket ins Projekt holen kann. In diesem Fall nutzen wir die Bibliothek iTextSharp. Diese kann noch einiges mehr, aber wir wollen erst einmal nur PDF-Dokumente zusammenführen.

iTextSharp

iTextSharp ist eine C#-Portierung des Projekts **iText 5**, für das Sie unter dem folgenden Link eine Reihe von Beispielen in der Programmiersprache Java finden: **http://developers.itextpdf.com/examples-itext5**. Mit **iText 5** können Sie eine ganze Menge Aufgaben rund um die Erstellung und Bearbeitung von PDF-Dokumenten erledigen. Aktuell wollen wir jedoch nur zwei oder mehrere PDF-Dokumente zusammenführen.

Beispielanwendung

Um die Bibliothek **iTextSharp** in ein Projekt zu integrieren, nutzen Sie den NuGet-Manager. Ausgehend von einem geöffneten Projekt klicken Sie dann mit der rechten Maustaste auf das Projekt im Projektmappen-Explorer. Aus dem Kontextmenü wählen Sie dann den Eintrag **NuGet-Pakete verwalten ...** aus. Im nun erscheinenden NuGet-Paket-Manager wechseln Sie zum Bereich Durchsuchen und geben als Suchbegriff **iTextSharp** ein. Ganz oben in der Liste erscheint nun der Eintrag **iTextSharp**, den Sie auswählen. Klicken Sie dann auf Installieren, um das Paket zum Projekt hinzuzufügen (siehe Bild 1).

Unserem WPF-Beispielprojekt fügen wir nun auf der Seite **MainWindow.xaml** eine einfache Schaltfläche hinzu, mit der wir einen ersten Test starten wollen.

Wir gehen vereinfachend davon aus, dass wir einfach zwei Dokumente namens **1.pdf** und **2.pdf** zusammenführen wollen.

Dazu fügen wir diese zum Projekt hinzu und stellen in den Eigenschaften dieser beiden Elemente den Wert **Immer kopieren** für die Eigenschaft **In Ausgabeverzeichnis kopieren** ein. So können wir das Verzeichnis für diese beiden

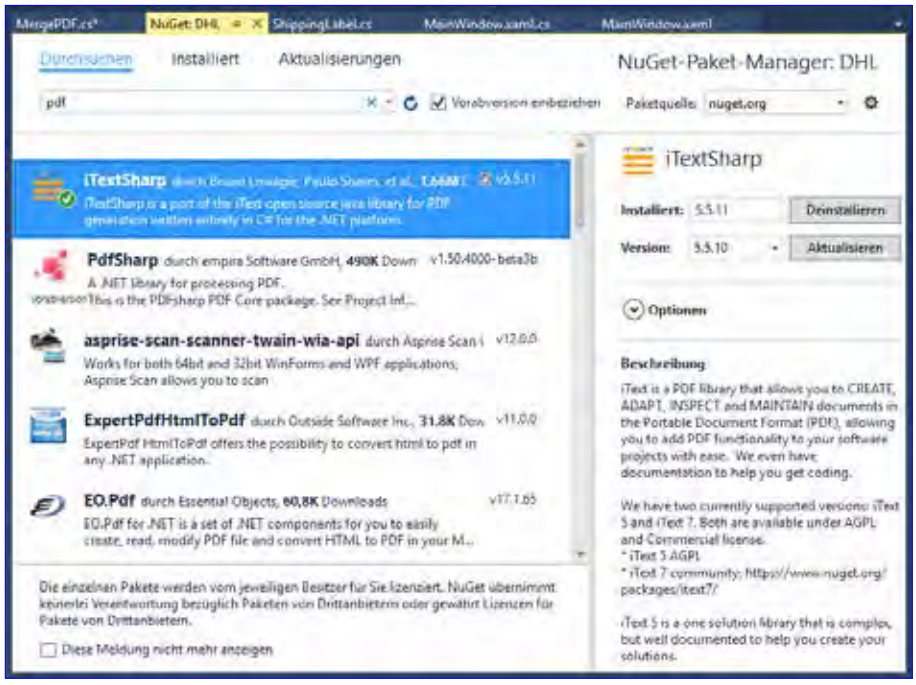


Bild 1: Installieren des NuGet-Pakets **iTextSharp**

Dokumente gleich direkt über die Funktion **AppDomain.CurrentDomain.BaseDirectory** ermitteln (dies liefert das Verzeichnis, in dem sich auch die **.exe**-Datei der Lösung befindet).

Dokumente zusammenführen

Wir wollen nun eine Methode programmieren, welche diese beiden Dokumente zusammenführt und als neues PDF-Dokument abspeichert. Das geht mit der **iTextSharp**-Bibliothek wirklich sehr einfach. Für die Schaltfläche hinterlegen wir die folgende Methode, die zunächst drei Pfade in den Variablen **document1**, **document2** und **target** hinterlegt – jeweils mit dem aktuelle Anwendungsverzeichnis und den Namen **1.pdf**, **2.pdf** und **12.pdf**:

```
private void btnTestZweiDokumente_Click(object sender, RoutedEventArgs e) {
    string document1 = AppDomain.CurrentDomain.BaseDirectory + "\\1.pdf";
    string document2 = AppDomain.CurrentDomain.BaseDirectory + "\\2.pdf";
    string target = AppDomain.CurrentDomain.BaseDirectory + "\\12.pdf";
```

Dann erstellt die Methode ein neues Objekt des Typs **Document** und eines vom Typ **FileStream**. Letzteres wird auf Basis des Pfades zur Zieldatei als neue Datei angelegt:

```
    Document document = new Document();
    FileStream stream = new FileStream(target, FileMode.Create);
```

Das Objekt **pdf** mit dem Typ **PdfCopy** führt das **Document**- und das **FileStream**-Objekt zusammen. Ein Objekt des Typs **PdfReader**, das später die Ausgangsdokumente aufnimmt, wird mit dem Wert **null** initialisiert. Dann wird das **Document**-Objekt mit der **open**-Methode geöffnet:

```
    PdfCopy pdf = new PdfCopy(document, stream);
    PdfReader reader = null;
    document.Open();
```

Nun folgen die beiden Einlesevorgänge der zusammenzuführenden Dokumente. Dabei wird **reader** jeweils mit einem neuen **PdfReader**-Objekt mit dem Dokumentpfad als Konstruktor-Parameter initialisiert und für die Anwendung innerhalb des **using**-Konstrukts vorbereitet. Darin wird das Dokument aus der Objektvariablen **reader** zum **pdfCopy**-Objekt hinzugefügt:

```
    using (reader = new PdfReader(document1)) {
        pdf.AddDocument(reader);
    }
```

Das erledigen wir auch noch für das zweite Dokument, also **2.pdf**:

```
    using (reader = new PdfReader(document2)) {
        pdf.AddDocument(reader);
    }
```

Schließlich schließt die Methode das Objekt **document** mit der **Close**-Methode:

```
    document.Close();
}
```

Das Ergebnis überzeugt: Beide Dokumente erscheinen im neuen Dokument **12.pdf**.

Anwendung bauen

Wenn wir nun schon so eine einfache Möglichkeit haben, PDF-Dokumente zusammenzustellen, wollen wir auch gleich eine praktische Anwendung daraus bauen. Diese soll es ermöglichen, über einen Datei auswählen-Dialog verschiedene PDF-Dokumente auszuwählen und diese so einer Liste hinzuzufügen. Ein Mausklick soll die Dokumente dann unter einem ebenfalls per Dateidialog auszuwählenden Dateinamen zusammenführen und speichern.

Um dies zu realisieren, benötigen wir ein Listenfeld, dass die Dateien anzeigt, sowie eine Schaltfläche zum Hinzufügen der gewünschten Dateien. Außerdem brauchen wir noch eine Schaltfläche, die den Datei speichern-Dialog anzeigt und nach der Eingabe eines entsprechenden Dateinamens die Dateien in der Reihenfolge wie im Listenfeld zusammenzuführen. Interessant wäre es natürlich noch, die Dateien im Listenfeld nach oben oder unten verschieben zu können. Dies wollen wir jedoch hintenanstellen und uns zunächst um die grundlegenden Funktionen kümmern.

Für diese Lösung verlassen wir das Beispielprojekt und erstellen ein neues WPF-Projekt namens **PDFMerger**. Diesem fügen Sie nun zunächst wieder das NuGet-Paket **iTextSharp** hinzu. Dann legen wir die benötigten Steuerelemente im Fenster **MainWindow.xaml** an. Der dazu verwendete Code sieht wie folgt aus. Der **<Windows.Resources>**-Teil legt zwei allgemeine Eigenschaften für die enthaltenen **Button**-Elemente fest:

```
<Window x:Class="PDFMerger.MainWindow" ... Title="MainWindow" Height="350" Width="525">
    <Window.Resources>
        <Style TargetType="{x:Type Button}">
            <Setter Property="Margin" Value="5"></Setter>
            <Setter Property="Height" Value="25"></Setter>
        </Style>
    </Window.Resources>
```

Das Grid besteht aus zwei Zeilen, von denen die obere das **ListView**-Element und das untere ein horizontal orientiertes StackPanel enthält, welches wiederum die fünf Schaltflächen aufnimmt:

```
    <Grid>
        <Grid.RowDefinitions>
            <RowDefinition></RowDefinition>
            <RowDefinition Height="Auto"></RowDefinition>
        </Grid.RowDefinitions>
        <ListView x:Name="lstDateien" Margin="5"></ListView>
```

```
        <StackPanel Orientation="Horizontal" Grid.Row="1">
            <Button x:Name="btnDateiAuswaehlen" Click="btnDateiAuswaehlen_Click">Dateien auswählen</Button>
            <Button x:Name="btnDateienZusammenfuehren" Click="btnDateienZusammenfuehren_Click">Dateien zusammenführen</Button>
            <Button x:Name="btnListeLeeren" Click="btnListeLeeren_Click">Liste leeren</Button>
            <Button x:Name="btnAuf" Click="btnAuf_Click">Auf</Button>
            <Button x:Name="btnAb" Click="btnAb_Click">Ab</Button>
        </StackPanel>
    </Grid>
</Window>
```

Im Entwurf sieht das Fenster nun wie in Bild 2 aus.

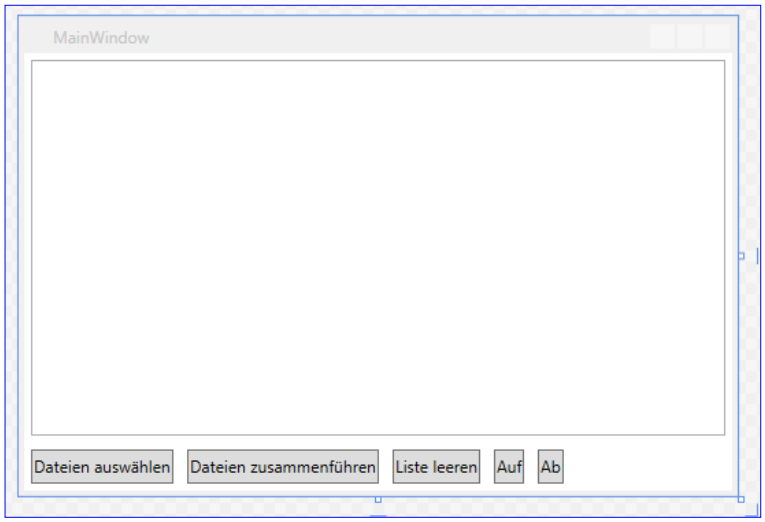


Bild 2: Entwurf des Fensters zum Mergen von PDF-Dokumenten

Fenster programmieren

Die Code behind-Klasse **MainWindow.xaml.cs** verwendet die folgenden Namespaces:

```
using System;
using System.Windows;
using Microsoft.Win32;
using iTextSharp.text;
using iTextSharp.text.pdf;
using System.IO;
using System.Collections.ObjectModel;
```

Die per Dateidialog ermittelten Namen der zusammenzuführenden PDF-Dateien sollen in einer **ObservableCollection** namens **Dateien** aufgenommen werden, die wir als private Variable deklarieren und initialisieren:

```
private ObservableCollection<string> Dateien = new ObservableCollection<string>();
```

Damit die in der **ObservableCollection** enthaltenen Dateinamen im **ListView**-Steuerelement angezeigt werden, weisen wir der Eigenschaft ItemsSource in der Konstruktor-Methode der Klasse einen Verweis auf die Collection zu:

```
public MainWindow() {
    InitializeComponent();
    lstDateien.ItemsSource = this.Dateien;
}
```

Ein Klick auf die Schaltfläche **btnDateiAuswaehlen** ruft die folgende Ereignismethode auf. Die Methode erstellt ein neues Objekt des Typs **OpenFileDialog** und stellt einige Eigenschaften für die entsprechende Objektvariable ein. Dieses erlaubt so die Auswahl mehrerer Dateien gleichzeitig (**Multiselect = true**), zeigt nur **.pdf**-Dokumente an (**Filter = "PDF-Dokumente (*.pdf)|*.pdf"**) und startet mit dem aktuelle Anwendungsverzeichnis:

EDM: Bestellungen und Bestellpositionen

Auch in dieser Ausgabe geht es mit unserer Beispiellösung Bestellverwaltung weiter. Nachdem wir bereits Kunden, Produkte und Kategorien verwalten können, nehmen wir uns nun die Bestellungen und Bestellpositionen vor. Dabei brauchen wir ein paar neue Seiten, um die Details einer Bestellung mit den Bestellpositionen und die Details einer neuen oder zu bearbeitenden Bestellposition anzuzeigen. Damit werden wir erstmal die Bearbeitung von Daten einer m:n-Beziehung ermöglichen.

Neue Funktionen

Die neue Version der Bestellverwaltung enthält wieder eine neue Gruppe im Ribbon, mit der zwei neue Seiten geöffnet werden können. Die erste davon ist eine Übersicht der Bestellungen in absteigender Reihenfolge nach dem Bestelldatum (siehe Bild 1).

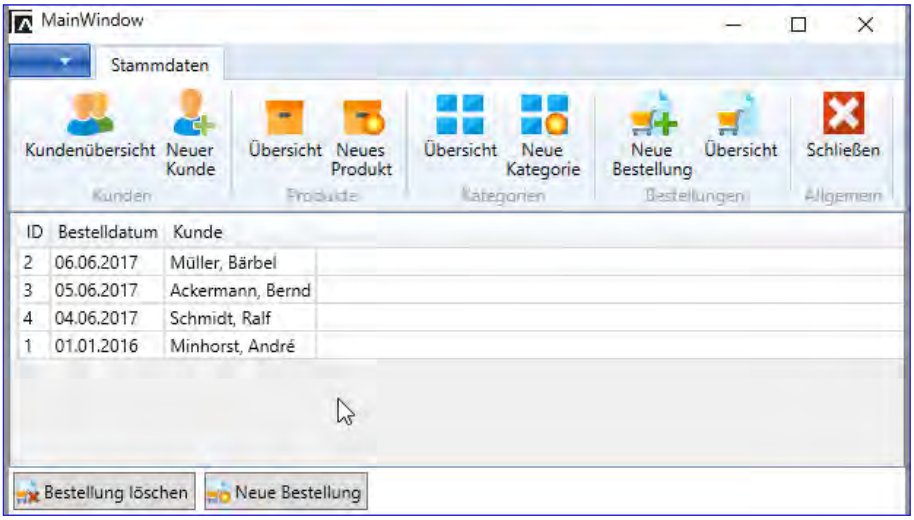


Bild 1: Übersicht der Bestellungen

Hier kann der Benutzer die aktuell markierte Bestellung per Mausklick löschen oder auch eine neue Bestellung anlegen. Per Doppelklick lässt sich die angeklickte Bestellung öffnen (siehe Bild 2).

Ein Klick auf den Ribbon-Eintrag **Neue Bestellung** soll den Dialog mit den Bestelldetails für eine neue Bestellung anzeigen, wobei direkt das Kombinationsfeld zur Auswahl des Kunden für diese Bestellung ausgeklappt wird (siehe Bild 3).

Auch wenn der Benutzer mit einem Klick auf die Schaltfläche **Neue Bestellposition** klickt, soll sich direkt das Kombinationsfeld zur Auswahl des betroffenen Produkts öffnen und die schnelle Auswahl erleichtern.

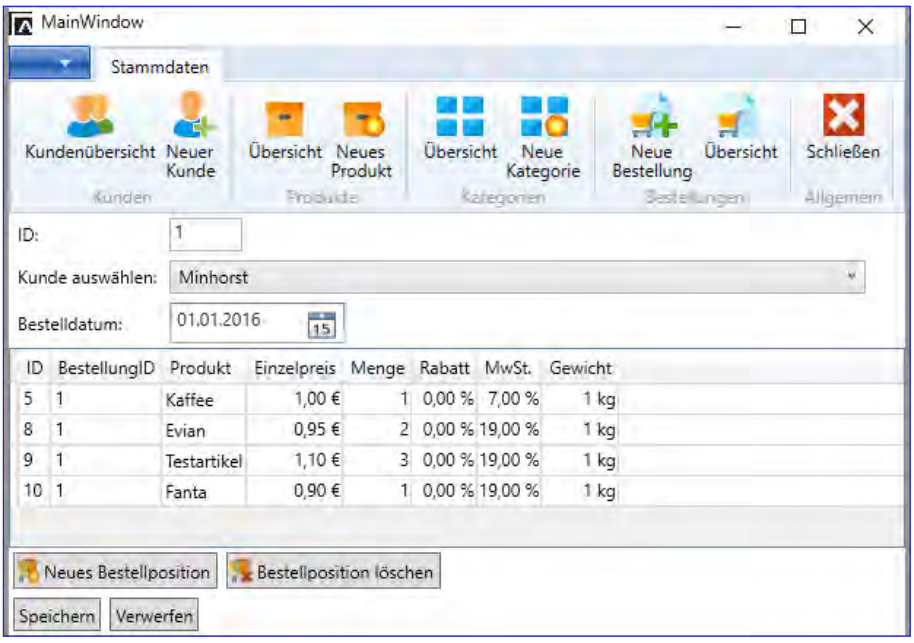


Bild 2: Bestellung in der Detailansicht

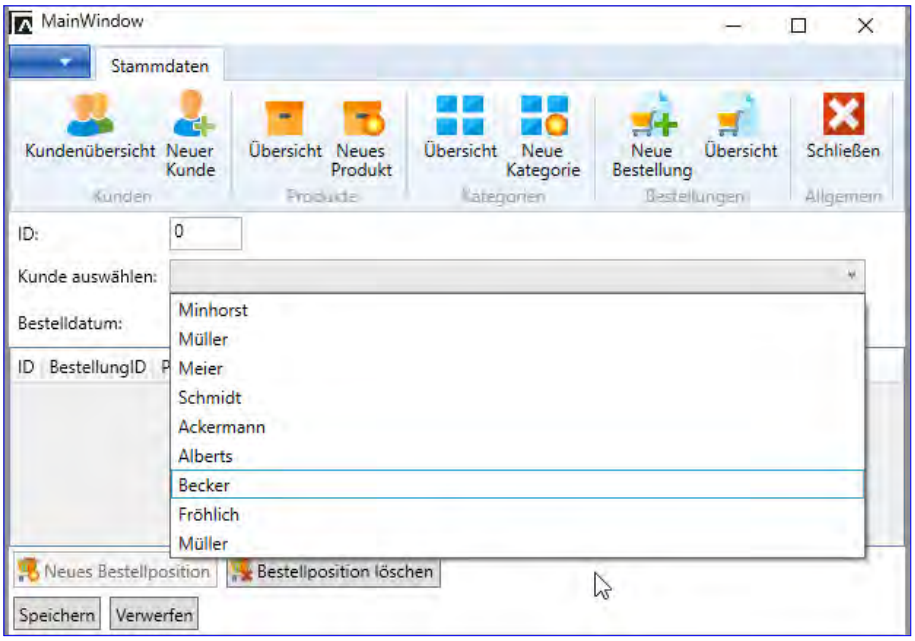


Bild 3: Anlegen einer neuen Bestellung

Wenn der Benutzer dann das gewünschte Produkte ausgewählt hat, sollen direkt die Werte der Felder **Einzelpreis**, **Mehrwertsteuersatz** und **Gewicht** aus der Tabelle **Produkte** in den neuen Datensatz der Tabelle **tblBestellpositionen** übernommen werden.

Die Detailseite für eine Bestellposition, wie sie nach einem Doppelklick auf einen der Einträge der Seite mit den Bestelldetails aussieht, finden Sie in Bild 4.

Datenbankerweiterungen

Für diese neue Version der Bestellverwaltung haben wir ein paar kleine Änderungen am Datenmodell der SQLite-Datenbank **Bestellungen.db** vorgenommen. Dabei sind wie auf zwei Probleme gestoßen, die wir wie folgt gelöst haben:

- Mit dem **DBBrowser for SQLite** konnten wir in manchen Fällen keine Zahlenfelder zu einer bestehenden Tabelle hinzufügen. Dies führt im schlimmsten Fall zum Leeren der kompletten Datenbank. Wir sind dann zu einem anderen Tool namens **SQLiteStudio** gewechselt (**sqlitestudio.pl**), welches die Änderungen klaglos vorgenommen hat.

- Als wir die Änderungen an der Datenbank in das Entity Data Model übernehmen wollten, gab es ebenfalls Probleme. Die Änderungen übernehmen Sie, indem Sie das Entity Data Model (Datei **BestellverwaltungEntities.edmx**) öffnen und dann den Kontextmenüeintrag **Modell aus der Datenbank aktualisieren ...** auswählen). Die nachfolgende Fehlermeldung, deren Text mit **[A]System.Data.SQLite.SQLiteConnection cannot be cast to [B]System.Data.SQLite.SQLiteConnection.** begann, wurde behoben, nachdem wir die aktuelle Version des NuGet-Pakets **System.Data.SQLite** sowie die aktuelle Runtime-Komponente

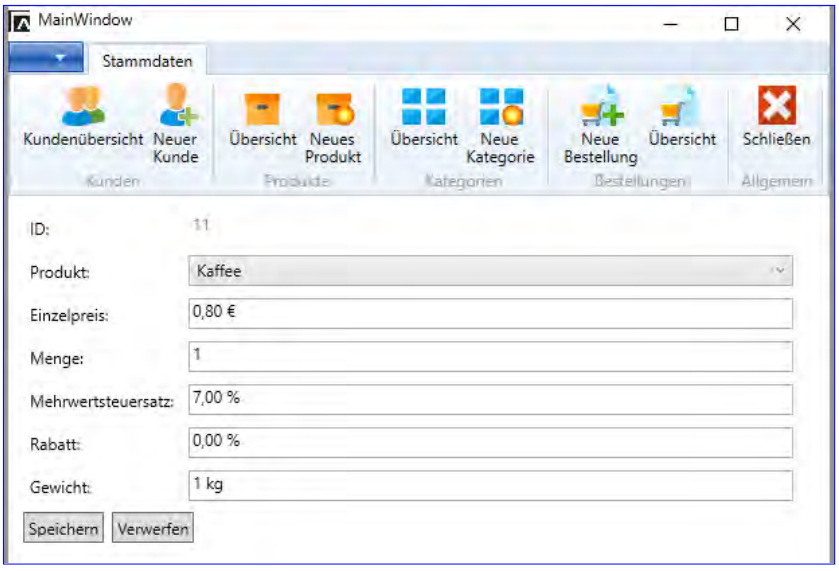


Bild 4: Bearbeiten einer Bestellposition

von der Internetseite **https://system.data.sqlite.org/index.html/doc/trunk/www/downloads.wiki** installiert und Visual Studio dann neu gestartet haben.

Hinzufügen der neuen Ribbon-Elemente

Dieser Teil bringt keine neuen Techniken. Die beiden neu hinzugefügten Ribbon-Schaltflächen heißen **btnNeueBestellung** und **btnBestelluebersicht** und laden die Seiten **BestellungDetails.xaml** beziehungsweise **BestellungUebersicht.xaml** in das **Frame**-Element namens **Workzone**.

Die Schaltfläche **btnBestelluebersicht** löst die folgende Ereignismethode aus:

```
private void btnBestelluebersicht_Click(object sender, RoutedEventArgs e) {
    ShowBestellungenUebersicht();
}
```

Die hier aufgerufene Methode **ShowBestellungenUebersicht** verwendet den optinalen Parameter **bestellungID**, der beim Aufruf nach dem Ändern oder Neuanlegen einer Bestellung übergeben werden kann, damit der geänderte oder neu angelegte Eintrag im DataGrid markiert werden kann.

Wurde die Seite **BestellungUebersicht** noch nicht erstellt oder gab es eine Änderung bei einer der Bestellungen, wird die Seite neu erstellt und die Variable **BestellungChanged** auf **false** eingestellt. Anschließend abonniert die Klasse **MainWindow.xaml.cs** die beiden Ereignisse **ItemDoubleClicked** und **NewItem** der Klasse **BestellungUebersicht.xaml.cs**. Schließlich wird die Seite, egal ob neu erzeugt oder bereits vorhanden, im **Frame**-Steuerelement **Workzone** angezeigt:

```
private void ShowBestellungenUebersicht(long bestellungID = 0) {
    if (bestellungUebersicht == null | BestellungChanged == true) {
        bestellungUebersicht = new BestellungUebersicht(bestellungID);
        BestellungChanged = false;
    }
    bestellungUebersicht.ItemDoubleClicked += new BestellungUebersicht.EventHandlerItemDoubleClicked(OnBestellungDoubleCl
icked);
    bestellungUebersicht.NewItem += new BestellungUebersicht.EventHandlerNewItem(OnNewBestellung);
    WorkZone.Content = bestellungUebersicht;
}
```

Übersicht der Bestellungen

Schauen wir uns erst die Seite **BestellungUebersicht.xaml** an. Diese enthält unter anderem ein DataGrid mit drei Spalten, die wie folgt definiert werden. Interessant ist hier wieder die in eine Resource des **DataGrid**-Elements ausgelagerte Ereignisdefinition für das Ereignis **MouseDoubleClick**, welches die Methode **Row_DoubleClick** aufrufen soll.

Außerdem haben wir hier, um die beiden Felder **Nachname** und **Vorname** der Auflistung **Kunden** des **Bestellung**-Objekts in einem **MultiBinding**-Element zusammengefasst:

```
<DataGrid x:Name="dgBestellungen" ItemsSource="{Binding Bestellungen}" AutoGenerateColumns="false" CanUserAddRows="False"
Grid.Row="0">
    <DataGrid.Columns>
        <DataGridTextColumn Binding="{Binding Path=ID}" Header="ID"  />
        <DataGridTextColumn Binding="{Binding Path=Bestelldatum, StringFormat=\{0:dd.MM.yyyy\}}" Header="Bestelldatum" />
        <DataGridTextColumn Header="Kunde Multibinding">
            <DataGridTextColumn.Binding>
                <MultiBinding StringFormat="{}{0}, {1}">
                    <Binding Path="Kunden.Nachname" />
                    <Binding Path="Kunden.Vorname" />
                </MultiBinding>
            </DataGridTextColumn.Binding>
        </DataGridTextColumn>
        <DataGridTextColumn Binding="{Binding Path=Kunde}" Header="Kunde" />
    </DataGrid.Columns>
    <DataGrid.Resources>
        <Style TargetType="DataGridRow">
            <EventSetter Event="MouseDoubleClick" Handler="Row_DoubleClick"/>
        </Style>
    </DataGrid.Resources>
</DataGrid>
```

Das DataGrid wird mit den Bestellungen aus einer **ObservableCollection** namens **bestellungen** mit Elementen des Typs **Bestellung** gefüttert, und zwar über die öffentliche Eigenschaft **Bestellungen**:

```
private ObservableCollection<Bestellung> bestellungen;
public ObservableCollection<Bestellung> Bestellungen {
    get { return bestellungen; }
    set { bestellungen = value; }
}
```

In der Konstruktor-Methode **BestellungUebersicht** erstellen wir das **dbContext**-Element und füllen die Collection **bestellungen**:

```
public BestellungUebersicht(long bestellungID = 0) {
    InitializeComponent();
    dbContext = new BestellverwaltungEntities();
    bestellungen = new ObservableCollection<Bestellung>(dbContext.Bestellungen.OrderByDescending(c => c.Bestelldatum));
    DataContext = this;
    if (bestellungID != 0) {
        Bestellung currentBestellung = dbContext.Bestellungen.Find(bestellungID);
```

```
            dgBestellungen.Focus();
            dgBestellungen.SelectedItem = currentBestellung;
            dgBestellungen.ScrollIntoView(currentBestellung);
        }
}
```

Neue Bestellung anlegen

Ein Klick auf die Schaltfläche **btnNeu** der Seite **BestellungUebersicht.xaml** löst die folgende Methode aus, welche wiederum das Ereignis **NewItem** aufruft:

```
private void btnNeu_Click(object sender, RoutedEventArgs e) {
    NewItem(this, null);
}
```

Bild 5: Entwurf der Bestelldetails samt DataGrid für die Bestellpositionen

NewItem ist wie folgt in **BestellungUebersicht.xaml.cs** definiert:

```
public delegate void EventHandlerNewItem(object sender, System.EventArgs e);
public event EventHandlerNewItem NewItem;
```

Das Ereignis **NewItem** wird im übergeordneten Fenster-Element **MainWindow.xaml** abonniert, sobald der Benutzer auf die Ribbon-Schaltfläche **btnBestelluebersicht** klickt (mehr siehe in den Methoden **btnBestelluebersicht_Click** und **ShowBestellungenUebersicht** in der Datei **MainWindow.xaml.cs**). Das Ereignis wird in **MainWindow.xaml.cs** durch die folgende Ereignisprozedur implementiert:

```
private void OnNewBestellung(object sender, EventArgs e) {
    BestelldetailsAnzeigen();
}
```

Damit steigen wir in die Methode **BestelldetailsAnzeigen** ein, die wir ebenfalls in der Klasse **MainWindow.xaml.cs** finden. Diese erstellt ein neues Objekt auf Basis der Seite **BestelldetailsAnzeigen** und übergibt den Parameter **bestellungID**, der hier **0** ist. Dann abonniert sie vier Ereignisse, nämlich **ItemChanged**, **Cancelled**, **NewItem** und **ItemDoubleClicked**. Die ersten beiden sind für die beiden Schaltflächen **Speichern** und **Verwerfen**, die hinteren beiden für die Schaltflächen **Neue Bestellposition** und **Bestellposition löschen**, die zum DataGrid zur Anzeige der Bestellpositionen gehören (siehe Bild 5). Nach dem Abonnieren der Ereignisse weist die Methode das neu erstellte **Page**-Objekt auf Basis der Klasse **BestellungDetails** noch der Eigenschaft **Content** des **Frame**-Objekts **Workzone** zu, welches die Seite dann anzeigt:

```
private void BestelldetailsAnzeigen(long bestellungID = 0) {
    bestellungDetails = new BestellungDetails(bestellungID);
    bestellungDetails.ItemChanged += new BestellungDetails.EventHandlerItemChanged(OnBestellungChanged);
    bestellungDetails.Cancelled += new BestellungDetails.EventHandlerCancelled(OnBestellungCancelled);
```

```
    bestellungDetails.NewItem += new BestellungDetails.EventHandlerNewItem(OnNewBestellposition);
    bestellungDetails.ItemDoubleClicked += new BestellungDetails.EventHandlerItemDoubleClicked(OnBestellpositionDoubleClicked);
    WorkZone.Content = bestellungDetails;
}
```

Felder in den Bestelldetails definieren

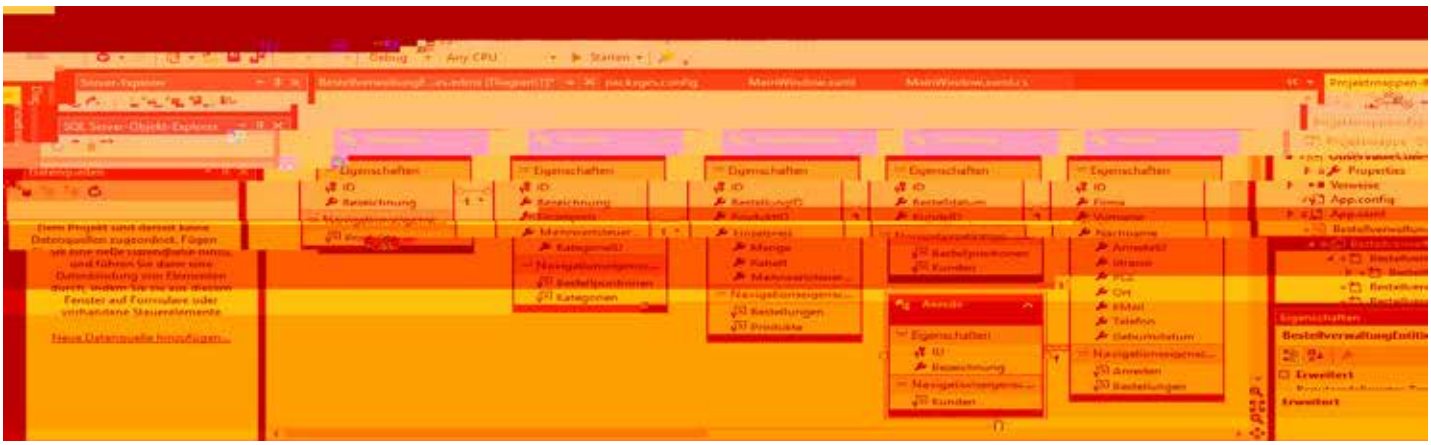
Die Seite **BestellungDetails.xaml** besteht aus drei Bereichen: den oberen Steuerelementen zur Eingabe der Bestelldetails, dem DataGrid zur Anzeige der Bestellpositionen und den Schaltflächen zum Hinzufügen und Löschen von Bestellpositionen sowie den beiden Schaltflächen zum Speichern oder Verwerfen des aktuell angezeigten Objekts aus der Tabelle **Bestellungen**. Schauen wir uns zunächst die drei Steuerelemente an. Die TextBox mit dem Bezeichnungsfeld **ID** ist an das Feld **Bestellung.ID** gebunden. Das heißt, dass wir in der Code behind-Datei ein Objekt namens **Bestellung** bereitstellen müssen – dazu später mehr. Interessant ist die ComboBox. Hier haben wir nicht wie üblich nur ein einziges Feld als anzuzeigenden Inhalt angegeben, sondern einen aus den beiden Feldern **Nachname** und **Vorname** bestehenden Ausdruck, also etwa **Minhorst, André**. Dies erledigen wir, indem wir das Attribut **DisplayMemberPath** weglassen und stattdessen ein Unterelement namens **ComboBox.DataTemplate** einfügen. Dieses soll ein **TextBlock**-Element enthalten, dessen Eigenschaftselement **Text** wir mit einem Multibinding der beiden Felder **Nachname** und **Vorname** in der mit **StringFormat** angegebenen Formatierung ausgeben:

```
<Label Grid.Row="0">ID:</Label>
<TextBox Grid.Column="1" Width="50" HorizontalAlignment="Left"  Text="{Binding bestellung.ID, Mode=TwoWay,
ValidatesOnExceptions=true}" ></TextBox>
<Label Grid.Row="1">Kunde auswählen:</Label>
<ComboBox x:Name="cboKunden" Grid.Column="1" HorizontalAlignment="Stretch" TabIndex="0" Height="23" Grid.Row="1"
            ItemsSource="{Binding Kunden}"
            SelectedItem="{Binding bestellung.Kunden, ValidatesOnDataErrors=True}"
            SelectedValuePath="ID" SelectionChanged="cboKunden_SelectionChanged">
    <ComboBox.ItemTemplate>
        <DataTemplate>
            <TextBlock>
```

AUSGABE 03/2017

DATENBANK ENTWICKLER

MAGAZIN FÜR DIE DATENBANKENTWICKLUNG MIT VISUAL STUDIO FÜR DESKTOP, WEB UND CO.



TOP-THEMEN:

WPF-CONTROLS	Das ListBox-Steuerelement: Grundlagen	SEITE 3
WPF-BASICS	EDM: Datenbindung mit dem ListBox-Steuerelement	SEITE 12
WPF-BASICS	EDM: Bilder speichern und anzeigen	SEITE 20
DATENZUGRIFF	SQLite-Datenmodellierung per C#	SEITE 30
ANWENDUNGEN	Weitergabe von WPF/C#/SQLite-Anwendungen	SEITE 53

Das ListBox-Steuerelement: Grundlagen

Wer unter Access/VBA gern das Listenfeld als Steuerelement genutzt hat, wird unter WPF ein ähnliches Steuerelement verwenden wollen. Ein einfach aufgebautes Steuerelement zur Anzeige von Texten in Listenform ist das ListBox-Steuerelement. Es lässt sich sehr leicht definieren und mit Daten füllen, sei es vordefiniert per XAML oder auch zur Laufzeit per Code. Dieser Artikel zeigt die Grundlagen zur Definition und Programmierung des ListBox-Steuerelements. Dabei erfahren Sie, wie Sie es mit einfachen Texten füllen, Einträge selektieren, die Selektion auslesen und auf Ereignisse wie die geänderte Auswahl oder einen Doppelklick reagieren können.

Programmiersprache

Die Beispiele in diesem Artikel basieren auf C#.

Hinweis

Um die verschiedenen Beispiele übersichtlich darzustellen, haben wir diese statt in verschiedenen **Window**-Elementen in **Page**-Elementen untergebracht, die dann über das Ribbon im Kopf des Fensters **MainWindow** per Mausklick auf den jeweiligen Ribbon-Button geöffnet werden können.

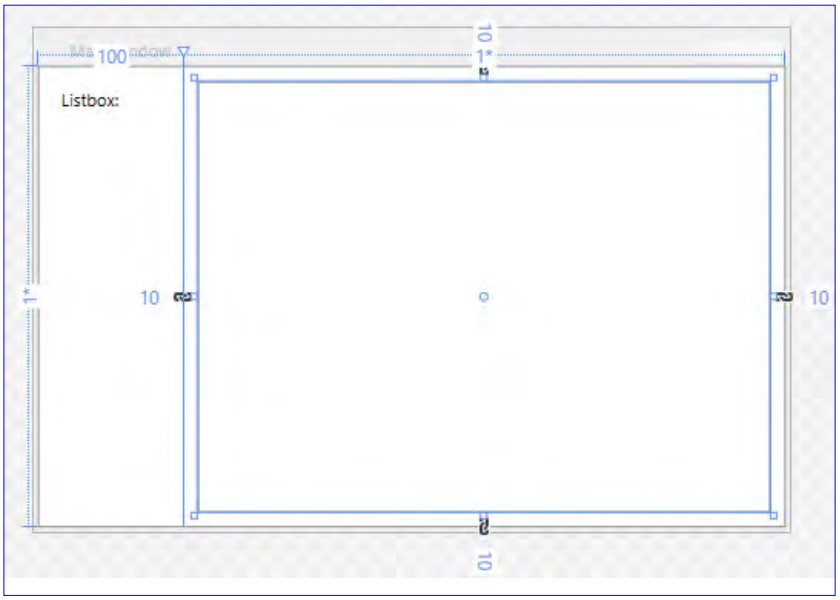


Bild 1: Entwurf unseres **ListBox**-Steuerelements

Die Definition des **RibbonButton**-Elements enthält dann im Attribut **tag** den Namen des zu öffnenden **Page**-Elements:

```
<RibbonButton Label="Einfache Einträge" Click="RibbonButton_Click" Tag="pgeEinfacheEintraege"></RibbonButton>
```

Die dort angegebene Ereignismethode **Ribbon_Button_Click** fragt den Wert des **Tag**-Attributs ab und öffnet das entsprechende **Page**-Element im **Frame**-Steuerelement:

```
private void RibbonButton_Click(object sender, RoutedEventArgs e) {
    RibbonButton button = (RibbonButton)sender;
    string pge = button.Tag.ToString();
    Workzone.Content = GetInstance("ListBox_Beispiele." + pge);
}
```

Listbox hinzufügen

Der erste und einfachste Schritt ist das Hinzufügen des ListBox-Elements: Dazu ziehen Sie es einfach aus der Toolbox in das WPF-Fenster und lassen es dort fallen.

Oder Sie machen es wie ein richtiger Programmierer und fügen ein **ListBox**-Element zur XAML-Definition des Fensters hinzu:

```
<ListBox x:Name="lstBeispiel" Height="100" Width="300" Margin="10,10" HorizontalAlignment="Left"
VerticalAlignment="Top"></ListBox>
```

Zusammen mit einem Label soll unsere erste Beispiel-ListBox dann wie folgt deklariert sein und im Entwurf wie in Bild 1 aussehen:

```
<Grid>
    <Grid.RowDefinitions>
        <RowDefinition></RowDefinition>
    </Grid.RowDefinitions>
    <Grid.ColumnDefinitions>
        <ColumnDefinition Width="100"></ColumnDefinition>
        <ColumnDefinition></ColumnDefinition>
    </Grid.ColumnDefinitions>
    <Label Grid.Column="0" Margin="10,10,10,10">Listbox:</Label>
    <ListBox x:Name="lstBeispiel" Grid.Column="1" Margin="10,10,10,10"></ListBox>
</Grid>
```

Einträge vordefinieren

Vielleicht wissen Sie schon zur Entwurfszeit, welche Einträge die Liste anzeigen soll. Dann können Sie diese Elemente einfach per XAML festlegen. Der XAML-Code für die ListBox sieht dann wie folgt aus:

```
<ListBox x:Name="lstEintraege" Grid.Column="1" Margin="10,10,10,10">
    <ListBoxItem>Erster Eintrag</ListBoxItem>
    <ListBoxItem>Zweiter Eintrag</ListBoxItem>
    <ListBoxItem>Dritter Eintrag</ListBoxItem>
    <ListBoxItem>Vierter Eintrag</ListBoxItem>
</ListBox>
```

Diese Einträge werden dann in der ListBox wie in Bild 2 angezeigt.

Element per XAML vorselektieren

Wenn Sie die Elemente per XAML vordefinieren, können Sie auch einen der Einträge vorselektieren. Dazu fügen Sie dem betroffenen Element einfach das Attribut **IsSelected** hinzu und stellen es auf **true** ein:

```
<ListBoxItem IsSelected="true">Erster Eintrag</ListBoxItem>
```

ListBox-Element mit Texten füllen

Die erste Schaltfläche, die wir hinzufügen, soll einen Eintrag zum Listenfeld hinzufügen. Das erledigen wir mit der **Add**-Methode der **Items**-Auflistung der ListBox. Die Schaltfläche definieren wir wie folgt:

```
<Button x:Name="btnNamen" Grid.Column="0" Grid.
ColumnSpan="2" Grid.Row="1" Height="30" Mar-
gin="10,10,10,10" Click="btnNamen_Click">Namen</
Button>
```

Das **Click**-Ereignis soll diese Methode auslösen:

```
private void btnNeuerEintrag_Click(object sen-
der, RoutedEventArgs e) {
    lstEintraege.Items.Add("Fünfter Eintrag");
}
```

Das Ergebnis sieht schließlich wie in Bild 3 aus.

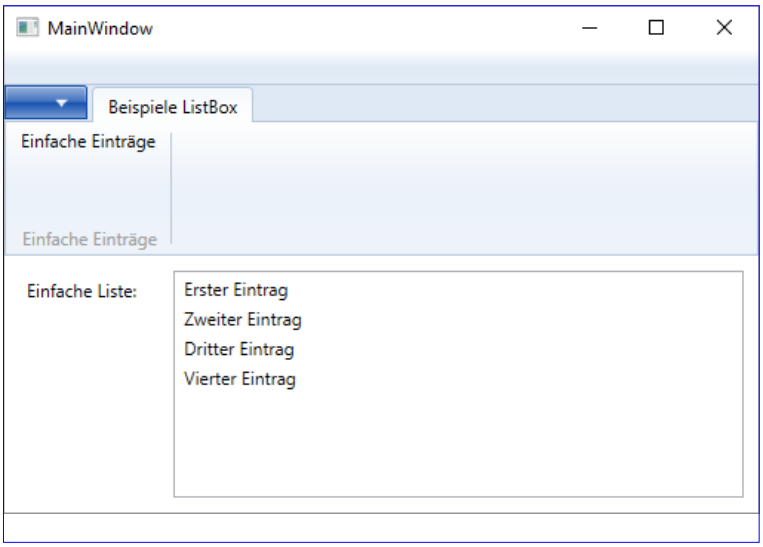


Bild 2: Feste Beispieleinträge in einem ListBox-Element

ListBox leeren

Wenn Sie alle Einträge aus der ListBox entfernen wollen, verwenden Sie die **Clear**-Methode der **Items**-Auflistung:

```
private void btnAlleLoeschen_Click(object sender, RoutedEventArgs e) {
    lstEintraege.Items.Clear();
}
```

Einfach- oder Mehrfachauswahl

Der Benutzer kann natürlich einen oder mehrere Einträge in der ListBox auswählen. Ob er einen oder mehrere auswählen kann, legen Sie mit der Eigenschaft **SelectionMode** fest. Diese kann die folgenden Werte annehmen:

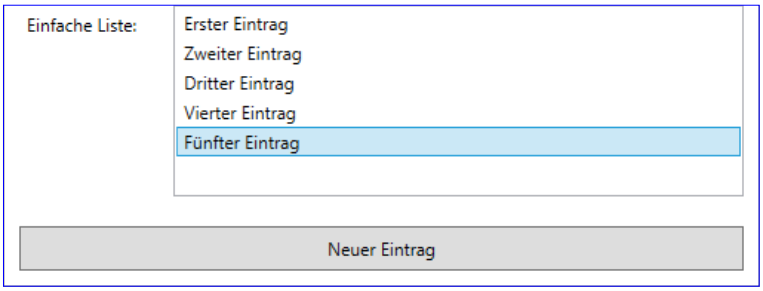


Bild 3: Einfügen einiger Einträge zu einem ListBox-Element

- **Single**: Standardeinstellung. Es kann nur ein Eintrag gleichzeitig ausgewählt sein. Allerdings kann der gewählte Eintrag auch nur noch abgewählt werden, indem man einen anderen Eintrag ausgewählt.

- **Multiple**: Der Benutzer kann einen, keinen oder mehrere Einträge auswählen. Zum Aus- oder Abwählen muss der jeweilige Eintrag angeklickt werden.

- **Extended**: Der Benutzer kann einen, keinen oder mehrere Einträge auswählen. Zum Aus- oder Abwählen gibt es mehrere Möglichkeiten: Ein einfacher Klick wählt einen Eintrag aus und alle anderen ab. Klicken bei gedrückter **Strg**-Taste arbeitet wie die Einstellung **Multiple**. Klicken bei gedrückter **Umschalt**-Taste wählt Bereiche aus.

Wir haben das in der Beispieldatenbank auf einer neuen Seite namens **pgeEintraegeAuswaehlen** abgebildet. Hier können Sie mit einer **ComboBox** einen der Modi für die Auswahl selektieren:

```
<ComboBox Grid.Column="1" Grid.Row="1" Margin="10,10,10,10" SelectionChanged="ComboBox_SelectionChanged">
    <ComboBoxItem IsSelected="true">Einfach</ComboBoxItem>
    <ComboBoxItem>Mehrfach</ComboBoxItem>
    <ComboBoxItem>Erweitert</ComboBoxItem>
</ComboBox>
```

Dies löst die folgende Ereignismethode aus:

```
private void ComboBox_SelectionChanged(object sender, SelectionChangedEventArgs e) {
    ComboBox cbo = (ComboBox)sender;
    switch (cbo.SelectedIndex) {
        case 0:
            lstEintraege.SelectionMode = SelectionMode.Single;
            break;
        case 1:
            lstEintraege.SelectionMode = SelectionMode.Multiple;
            break;
        case 2:
            lstEintraege.SelectionMode = SelectionMode.Extended;
            break;
        default:
            break;
    }
}
```

Das Ergebnis sehen Sie in Bild 4. Wenn Sie etwa den dritten Eintrag **Erweitert** auswählen, stellt dies die Option **Extended** ein. Der Benutzer kann dann sowohl bei gedrückter **Umschalt**-Taste ganze Bereiche auswählen, aber auch bei gedrückter **Strg**-Taste einzelne Einträge aus- oder abwählen – also ganz wie im Windows Explorer.

Gewählte Elemente abfragen

Wenn Sie den Benutzer Elemente auswählen lassen, wollen Sie auch abfragen, welche Elemente ausgewählt sind. Dazu gibt es verschiedene Möglichkeiten, die Sie auf der Seite **pgeAuswahlErmitteln** kennen lernen.

EDM: Datenbindung mit dem ListBox-Steuerelement

Im Artikel Das ListBox-Steuerelement: Grundlagen haben wir die Grundlagen zum ListBox-Steuerelement erläutert. Heute gehen wir einen Schritt weiter und zeigen, wie Sie die Daten aus Objekten anzeigen, die beispielsweise aus einer Datenbank-Tabelle gewonnen oder anderweitig in die Objekte gefüllt werden. Dabei zeigen wir auch, wie Sie in der ListBox anspruchsvollere Daten wie etwa mehrspaltige Daten oder auch Bilder anzeigen.

Voraussetzung

In diesem Artikel wollen wir zeigen, wie Sie die Daten aus einer SQLite-Datenbank über das Entity Data Model in einem ListBox-Steuerelement anzeigen und die verschiedenen Möglichkeiten dazu diskutieren. Dazu greifen wir auf das gleiche Entity Data Model zu, welches wir auch in der Beispielanwendung namens **Bestellverwaltung** nutzen.

ListBox mit Elementen füllen

Wenn Sie nur eine einfache ListBox benötigen, die lediglich etwa die Namen von Kategorien anzeigt, dann durchlaufen Sie per Code eine Liste der Kategorien und fügen die Einträge zur ListBox hinzu. Das Ergebnis soll wie in Bild 1 aussehen (siehe Beispielprojekt, Seite **pgeListeEinlesenUndZuweisen.xaml**). Die Definition des **ListBox**-Elements sieht dann einfach wie folgt aus:

```
<ListBox x:Name="lstKunden" Grid.Column="1" Margin="10,10,10,10" />
```

In der Code behind-Datei benötigen wir lediglich die folgenden Codezeilen:

```
BestellverwaltungEntities dbContext;
public pgeListeEinlesenUndZuweisen() {
    InitializeComponent();
    dbContext = new BestellverwaltungEntities();
    var produkte = from d in dbContext.Produkte select d;
    foreach(var produkt in produkte) {
        lstKunden.Items.Add(produkt.Bezeichnung);
    }
    DataContext = this;
}
```

Hier lesen wir mit einer LINQ-Abfrage über die Liste der Produkte alle Elemente ein. Diese speichern wir in der Auflistung produkte, deren Elemente wir dann in einer **foreach**-Schleife durchlaufen. In der **foreach**-Schleife fügen wir der ListBox dann

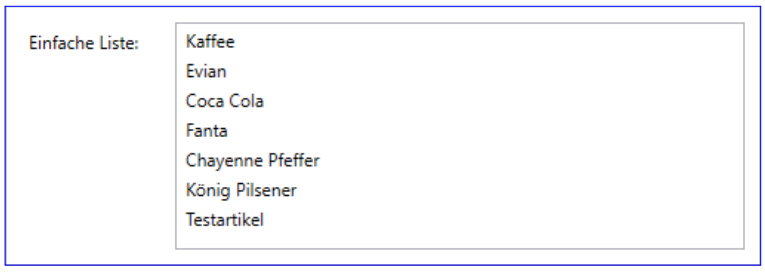


Bild 1: Anzeige der Bezeichnungen der Tabelle **Produkte**

mit der **Add**-Methode der **Items**-Auflistung die **Bezeichnung** des jeweiligen Produkts zu. Dieses Beispiel hilft uns natürlich nur weiter, wenn wir die Daten lediglich anzeigen und gegebenenfalls genau die angezeigten Werte weiterverwenden wollen. Wenn Sie über diese Einträge den Primärschlüsselwert des jeweiligen Datensatzes ermitteln wollen, ist es der falsche Ansatz. In diesem Fall benötigen Sie weitere Informationen.

ListBox an Liste binden

Im zweiten Beispiel (siehe Seite **pgeListeGebunden.xaml**) wollen wir das **ListBox**-Element an eine Liste binden, die wir aus dem Entity Data Model gefüllt haben. Die ListBox definieren wir im ersten Schritt wieder ganz einfach:

```
<ListBox x:Name="lstProdukte" Grid.Column="1" Margin="10,10,10,10" />
```

Der Code der Klasse sieht auch übersichtlich aus. Hier definieren wir zunächst den Datenbank-Kontext **dbContext**. Die Konstruktor-Methode erstellt den Datenbank-Kontext und füllt dann die Variable **produkte** mit einer Liste der Produkte, die wie wieder per LINQ-Abfrage ermitteln. Dann weisen wir die Liste **produkte** der Eigenschaft **ItemsSource** zu und stellen **DisplayMemberPath** auf das Feld **Bezeichnung** ein. Somit wird die Bezeichnung des jeweiligen Produkts angezeigt:

```
public partial class pgeListeGebunden : Page {
    BestellverwaltungEntities dbContext;
    public pgeListeGebunden() {
        InitializeComponent();
        dbContext = new BestellverwaltungEntities();
        var produkte = (from p in dbContext.Produkte select p).ToList();
        lstProdukte.ItemsSource = produkte;
        lstProdukte.DisplayMemberPath = "Bezeichnung";
    }
}
```

Das Ergebnis sieht genauso aus wie beim vorherigen Beispiel.

Binding im XAML-Code

Nun wollen wir noch ein wenig mehr mit den Bindungs-Eigenschaften der WPF-Steuerelemente arbeiten, sprich: die beiden Eigenschaften **ItemsSource** und **DisplayMemberPath** sollen im XAML-Code aufgeführt werden:

```
<ListBox x:Name="lstProdukte" ... ItemsSource="{Binding Produkte}" DisplayMemberPath="Bezeichnung"/>
```

Damit diese Eigenschaften auch die gewünschten Elemente in der Code behind-Datei finden, gestalten wir den Code dort etwas anders als beim vorhergehenden Beispiel. Nicht nur, dass wir die Zuweisung der beiden Eigenschaften **ItemsSource** und **DisplayMemberPath** dort natürlich entfernen.

Stattdessen fügen wir eine private Variable zum Speichern einer Produkte-Liste namens **produkte** hinzu. Diese machen wir über die öffentliche Eigenschaft **Produkte** von außen zugreifbar, also auch für die **ItemsSource**-Eigenschaft des **ListBox**-Elements:

```
public partial class pgeListeGebundenWPF : Page {
    BestellverwaltungEntities dbContext;
    private List<Produkt> produkte;
    public List<Produkt> Produkte {
        get { return produkte; }
        set { produkte = value; }
    }
```

Die Konstruktor-Methode muss dann natürlich dafür sorgen, dass **produkte** mit den gewünschte Einträgen gefüllt wird. Das erledigen wir, indem wir zuerst wieder den Datenbank-Kontext initialisieren und dann **produkte** mit einer Liste von **Produkt**-Elementen füllen, die wir per LINQ ermitteln:

```
    public pgeListeGebundenWPF() {
        InitializeComponent();
        dbContext = new BestellverwaltungEntities();
        produkte = new List<Produkt>((from p in dbContext.Produkte select p));
        DataContext = this;
    }
}
```

Eintrag hinzufügen

Wenn Sie der Liste etwa per Schaltfläche einen Eintrag hinzufügen wollen, sollte dieser natürlich auch direkt in der ListBox erscheinen. Einer neue Schaltfläche namens **btnNeuerEintrag** fügen wir etwa die folgende Methode für das Ereignis **Click** hinzu:

```
private void btnNeuerEintrag_Click(object sender, RoutedEventArgs e) {
    Produkt produkt = new Produkt();
    produkt.Bezeichnung = "Neues Produkt";
    produkte.Add(produkt);
}
```

Bei Anklicken wird so zwar nun ein neuer Eintrag zur Liste **produkte** hinzugefügt, was Sie durch Debuggen und Abfrage der **produkte**-Elemente prüfen können, allerdings erscheint der neue Eintrag nicht automatisch in der Liste. Das liegt daran, dass wir die Elemente in einer Auflistung des Typs **List** gespeichert haben. Sollen die Einträge nun aktualisiert werden, müssten Sie die Liste erneut der **ListBox** zuweisen.

Aktualisieren der ListBox mit einer ObservableCollection

Die Lösung ist das Austauschen der **List**-Auflistung durch eine **ObservableCollection**-Auflistung, wie wir es in **pgeListeGebundenWPFObservableCollection** gemacht haben.

Dort haben wir den Code der Code behind-Klasse wie folgt geändert. Zunächst haben wir den Typ der privaten und der öffentlichen Auflistung von **List** in **ObservableCollection** geändert:

```
public partial class pgeListeGebundenWPFObservableCollection : Page {
    BestellverwaltungEntities dbContext;
    private ObservableCollection<Produkt> produkte;
    public ObservableCollection<Produkt> Produkte {
        get { return produkte; }
        set { produkte = value; }
    }
```

Dementsprechend wird die Liste produkte mit einem neuen Objekt des Typs **ObservableCollection** gefüllt:

```
    public pgeListeGebundenWPFObservableCollection() {
        InitializeComponent();
        dbContext = new BestellverwaltungEntities();
        produkte = new ObservableCollection<Produkt>((from p in dbContext.Produkte select p));
        DataContext = this;
    }
```

Die Methode **btnNeuerEintrag_Click** arbeitet nun wie erwartet und fügt gleich den neuen Eintrag zur ListBox hinzu. Hier haben wir noch zwei Anweisungen hinzugefügt, die das neue **Produkt**-Element auch noch zur **Produkte**-Auflistung des Datenbank-Kontext hinzufügen und die Änderung speichern, damit diese auch in die Datenbank übertragen wird:

```
    private void btnNeuerEintrag_Click(object sender, RoutedEventArgs e) {
        Produkt produkt = new Produkt();
        produkt.Bezeichnung = "Neues Produkt";
        dbContext.Produkte.Add(produkt);
        produkte.Add(produkt);
        dbContext.SaveChanges();
    }
}
```

Einträge auswählen

Interessant wird es, wenn Sie die vom Benutzer getroffene Auswahl auswerten wollen. Das können Sie beispielsweise gleich nach der Auswahl erledigen oder auch zu einem späteren Zeitpunkt – etwa, wenn eine Schaltfläche betätigt wird.

Wir wollen uns im folgenden Beispiel (siehe **pgeAuswahlErmitteln**) zunächst das Ereignis **SelectionChanged** ansehen, das wir wie folgt im XAML-Code eintragen:

```
<ListBox x:Name="lstProdukte" Grid.Column="1" Margin="5" ItemsSource="{Binding Produkte}" DisplayMemberPath="Bezeichnung"
SelectionChanged="lstProdukte_SelectionChanged"/>
```

Nun benötigen wir noch die dadurch ausgelöste Methode. Diese definieren wir im Code behind-Modul wie folgt:

EDM: Bilder speichern und anzeigen

In der Bestellverwaltung haben wir bisher nur mit einfachen Daten in Datentypen wie Text, Zahl oder Datum gearbeitet. Dabei kann man doch mit WPF auch wunderbar Bilddateien anzeigen. In diesem Artikel erweitern wir die Datenbank Bestellverwaltung um das Speichern und die Anzeige von Bildern zu den Produktdatensätzen.

Speicherort: Tabellenfeld oder Dateisystem?

Eine der wichtigsten Fragen, wenn es darum geht, Bilder mit einer Datenbank zu verwalten, ist die nach dem Speicherort der Bilder. Grundsätzlich gibt es zwei einfache Varianten: Die Bilder werden im Dateisystem gespeichert und die Datenbank hält nur die Dateipfade zu den Bildern in den jeweiligen Datensätzen vor. Oder Sie speichern die Bilder direkt in einem entsprechenden Feld in der Datenbank. Wir werden an dieser Stelle, an der wir mit einer SQLite-Datenbank arbeiten, die Variante mit dem Dateisystem wählen, und zwar aus folgenden Gründen:

- Datenbanken sind dazu gemacht, referenziell verknüpfte Daten zu speichern, diese zu indizieren und schnell zugreifbar zu machen oder zu ändern. Das Speichern von Bildern oder anderen Dateien in den Tabellen der Datenbank ist da eher kontraproduktiv.

- Wenn Sie die Bilder in der Datenbankdatei speichern und die Daten sichern wollen, müssen Sie immer die komplette Datenbank inklusive der enthaltenen Bilder sichern. Wenn sich die Bilder im Dateisystem etwa in einem Ordner im gleichen Verzeichnis wie die Datenbank befinden, können diese separat gesichert werden, was nur dann nötig ist, wenn sich diese geändert haben.

- Zum Bearbeiten oder Anzeigen müssen die Bilder in der Regel entweder ohnehin im Dateisystem gespeichert oder aber anderweitig in den Speicher geladen werden. Dann kann man sie auch direkt aus dem Dateisytem heraus öffnen.

Optimalerweise speichern wir die Bilder auch noch in einem Verzeichnis, dass sich auf der gleichen Ebene oder unterhalb der Anwendung befindet. Auf diese Weise brauchen wir nur den relativen Pfad zur Bilddatei in der Datenbank zu speichern.

Tabelle Produkte erweitern

Damit wir die Bilder, die wir im Dateisystem speichern, mit den Datensätzen der Tabelle **Produkte** verknüpfen können, fügen wir dieser Tabelle ein Feld namens Bild hinzu. Es soll ein einfaches Textfeld sein, dass den Pfad zur Bilddatei speichert, und zwar relativ zur Datenbank in einem eigenen Ordner namens **Bilder**.

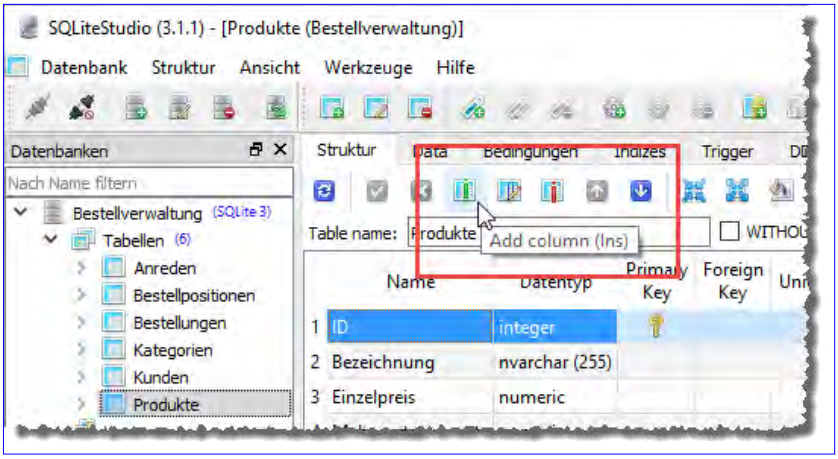


Bild 1: Anlegen eines neuen Feldes

Diese Änderung nehmen wir der Einfachheit halber über das Tool **SQLiteStudio** vor. Dort öffnen Sie die Datenbank **Bestellverwaltung** und klicken dann doppelt auf die Tabelle **Produkte**. Nun betätigen Sie die Schaltfläche **Add column (Ins)** wie in Bild 1.

Dies öffnet den Dialog aus Bild 2. Hier tragen Sie den Namen des neuen Feldes sowie den Datentyp ein. Anschließend schließen Sie den Dialog mit der **OK**-Schaltfläche.

Erst der Klick auf die Schaltfläche **Commit structure changes** öffnet den Dialog **Auszuführende Abfragen**, der die SQL-Abfragen mit den geplanten Änderungen anzeigt und per Klick auf **OK** die Änderung in die Datenbank überträgt (siehe Bild 3).

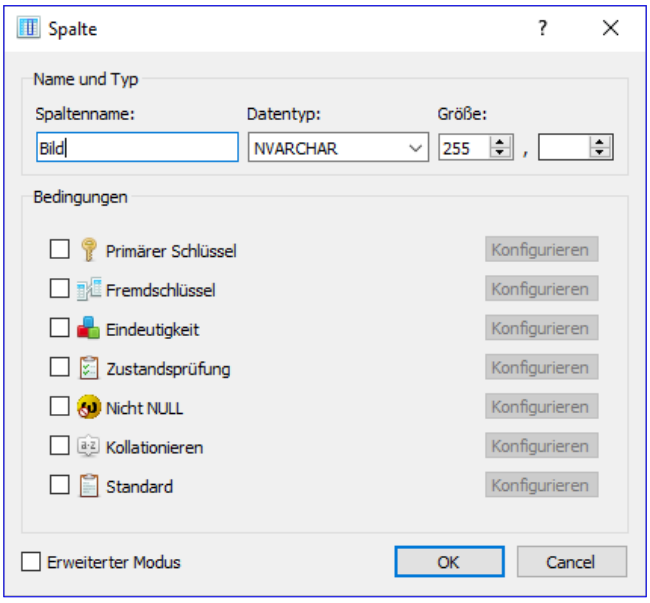


Bild 2: Anlegen des Feldes **Bild**

Nun müssen wir die Änderung in der Datenbank noch in das Entity Data Model übertragen. Dazu öffnen Sie die Ansicht **Bestellverwaltung.edmx** und wählen aus dem Kontextmenü den Eintrag **Modell aus der Datenbank aktualisieren...** aus (siehe Bild 4).

Danach erscheint das Feld **Bild** dann auch im Element **Produkt** des Entity Data Models. Nun können wir also Bildpfade zum Feld **Bild** des Elements **Produkt** hinzufügen. Damit sind die Arbeiten am Entity Framework erledigt.

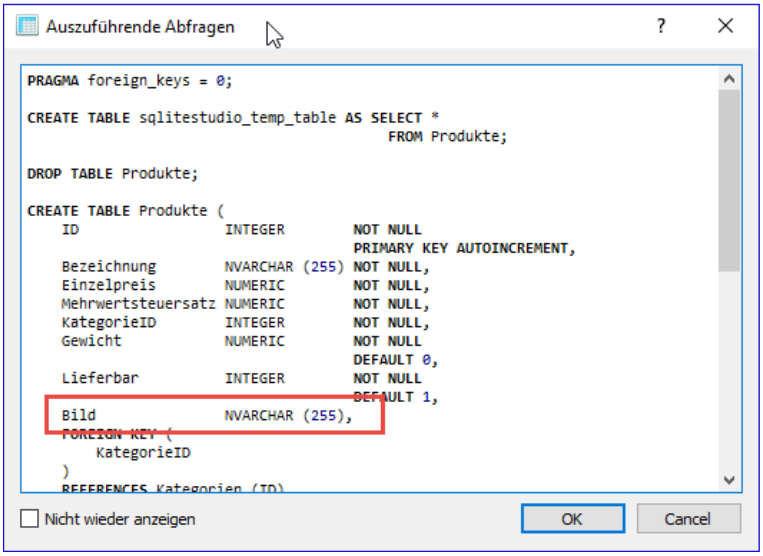


Bild 3: SQL-Anweisung zum Anlegen der Tabelle mit dem Feld **Bild**

XAML-Code definieren

Nun wollen wir das Aussehen des Fensters zur Anzeige der Produktdaten mit den Bildern definieren. Dies erledigen wir diesmal gleich im Fenster **MainWindow.XAML**. Damit die Steuerelemente wie gewünschten angeordnet werden, haben wir über die Elemente **Grid.RowDefinitions** und **Grid.ColumnDefinitions** einige Zeilen und Spalten definiert. Die Definition dieser Elemente sieht wie folgt aus, wobei die Position des Elements mit dem Wert * für das Attribut **Height** eine besondere Rolle spielt:

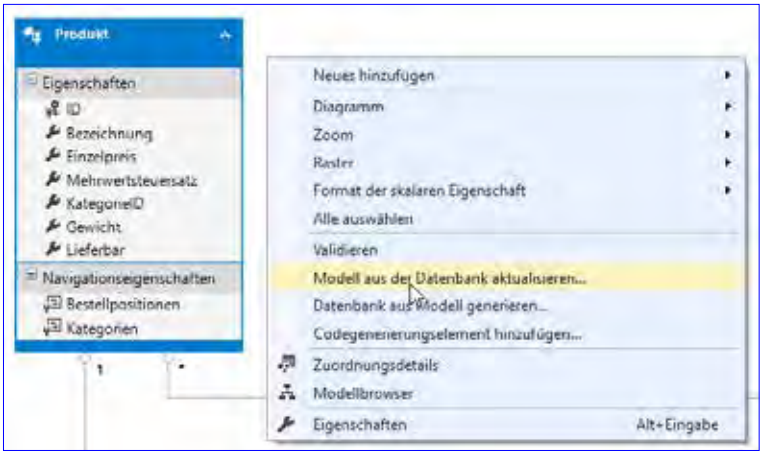


Bild 4: Aktualisieren des Entity Data Models

```
<Grid.RowDefinitions>
    <RowDefinition Height="Auto"></RowDefinition> //für Header
    <RowDefinition Height="Auto"></RowDefinition> //für ListBox
    <RowDefinition Height="Auto"></RowDefinition> //für ID des Produkts
    <RowDefinition Height="Auto"></RowDefinition> //für Bezeichnung
    <RowDefinition Height="Auto"></RowDefinition> //für Bildpfad
    <RowDefinition Height="*"></RowDefinition> //für Image-Steuerelement
    <RowDefinition Height="Auto"></RowDefinition> //für Schaltflächen
</Grid.RowDefinitions>
<Grid.ColumnDefinitions>
    <ColumnDefinition Width="Auto"></ColumnDefinition>
    <ColumnDefinition Width="*"></ColumnDefinition>
</Grid.ColumnDefinitions>
```

Hier hinterlegen wir nun das **ListBox**-Steuerelement, das in der zweiten Zeile über zwei Spalten angeordnet werden soll. Dieses fügen wir in der folgenden Form zum XAML-Code hinzu:

```
<ListBox x:Name="lstProdukte" Grid.Row="1" Grid.ColumnSpan="2" Margin="5" ItemsSource="{Binding Produkte}"
DisplayMemberPath="Bezeichnung" SelectionChanged="lstProdukte_SelectionChanged"/>
```

Das **ListBox**-Element heißt **lstProdukte** und ist an die Auflistung **Produkte** als **ItemsSource** gebunden. Angezeigt soll der Inhalt der Eigenschaft **Bezeichnung** des gebundenen Objekts. Wenn sich die Auswahl ändert, etwa weil der Benutzer ein anderes Element anklickt, soll dies die Ereignisprozedur **lstProdukte_SelectionChanged** auslösen. Damit das **ListBox**-Element die Elemente der Produkte auflistet, fügen wir der Code behind-Klasse **MainWindow.xaml.cs** einige Codezeilen hinzu. Den Beginn macht die klassenweite Definition des **DBContext**-Elements:

```
BestellverwaltungEntities dbContext;
```

Damit wir die Produkte als Datenherkunft für die ListBox haben und das einzelne Produkt-Objekt als Datenherkunft für die Steuerelemente des Fensters, definieren wir außerdem noch eine private Variable für die Liste der Produkte, und zwar für eine ObservableCollection mit Elementen des Typs **Produkt**:

```
private ObservableCollection<Produkt> produkte;
```

Den Inhalt dieser Variablen wollen wir auch öffentlich zugänglich machen, damit wir die Elemente des XAML-Codes daran binden können. Dies erledigen wir wie folgt:

```
public ObservableCollection<Produkt> Produkte {
    get { return produkte; }
    set { produkte = value; }
}
```

Auch für das jeweils mit den Textfeldern anzuzeigende Produkt, dessen Bild wir ja auch laden wollen, deklarieren wir ein Objekt namens **produkt**, dass eine Instanz des Typs **Produkt** speichern kann:

```
private Produkt produkt;
```

Die öffentliche Eigenschaft kann über den Namen **Produkt** gelesen und geschrieben werden. Damit beim Füllen der Variablen **produkt** gleich die daran gebundenen Steuerelemente aktualisiert werden, implemtieren wir die Schnittstelle **INotifyProperty-Changed** für die aktuelle Klasse. Der Teil in der öffentlichen Eigenschaft Produkt erhält dazu im Setter einen Aufruf der Methode **OnPropertyChanged**:

```
public Produkt Produkt {
    get { return produkt; }
    set {
        produkt = value;
        OnPropertyChanged(new PropertyChangedEventArgs("Produkt"));
    }
}
```

Für eine ordnungsgemäße Implementierung der Schnittstelle muss diese natürlich im Kopf der Klasse angegeben werden:

```
public partial class MainWindow : Window, INotifyPropertyChanged {
...
}
```

Außerdem benötigen wir noch diese beiden Elemente:

```
protected virtual void OnPropertyChanged(PropertyChangedEventArgs e) {
    if (PropertyChanged != null) {
        PropertyChanged(this, e);
    }
}
public event PropertyChangedEventHandler PropertyChanged;
```

Danach wenden wir uns der Konstruktor-Methode der Klasse zu. Diese initialisiert das Objekt auf Basis der XAML-Definition, initialisiert das DBContext-Objekt, füllt die ObservableCollection namens **produkte** und weist dem Fenster die Code behind-Klasse als Datenquelle zu. Damit das **ListBox**-Element das Element markiert, das beim Anzeigen des Fensters in den übrigen Steuerelementen angezeigt wird (und somit der privaten Variablen **produkt** zugewiesen ist), stellen wir produkt auf das mit **First** ermittelte erste Element der **produkte**-Auflistung ein und markieren gleich auch noch diesen Eintrag in der **ListBox**:

```
public MainWindow() {
    InitializeComponent();
```

SQLite-Datenmodellierung per C#

Wenn Sie eine SQLite-Datenbank von einer WPF/C#-Anwendung aus nutzen, möchten Sie gegebenenfalls einmal Tabellen in dieser Datenbank anlegen, ändern oder löschen oder gar neue Datenbanken mit C# erstellen. Dies kann beispielsweise interessant werden, wenn Sie eine neue Version einer Anwendung ausliefern, aber der Benutzer die Daten in der Datenbank der vorherigen Version weiter nutzen möchte. Dieser Artikel stellt die grundlegenden Vorgehensweisen für diese Arbeitsschritte vor.

NuGet-Paket für SQLite hinzufügen

Wenn Sie von einer C#-Anwendung aus Änderungen an einer SQLite-Datenbank vornehmen oder diese sogar erstellen wollen, benötigen Sie das NuGet-Paket mit den entsprechenden Erweiterungen. Dazu klicken Sie mit der rechten Maustaste auf das Projekt im Projektmappen-Explorer und wählen den Kontextmenü-Eintrag **NuGet-Pakete verwalten...** aus. Im nun erscheinenden Bereich wechseln Sie auf Durchsuchen und suchen nach **SQLite**. Den nun auftauchenden Eintrag **System.Data.SQLite** können Sie dann auswählen und installieren.

SQLite-Befehle verfügbar machen

In der Klasse, in der Sie die Befehle zum Erstellen und Anpassen von SQLite-Datenbanken unterbringen wollen, benötigen Sie zunächst einen Verweis auf den Namespace **System.Data.SQLite**, der erst nach dem Hinzufügen des oben genannten NuGet-Pakets vorfinden:

```
using System.Data.SQLite;
```

Leere Datenbank erstellen

Um eine leere Datenbank zu erstellen, fügen wir dem Fenster **MainWindow.xaml** ein Textfeld zur Eingabe des Datenbanknamens sowie eine Schaltfläche zum Erstellen der leeren Datenbankdatei hinzu. Diese Schaltfläche soll die folgende Methode auslösen:

```
private void btnCreateDatabase_Click(object sender, RoutedEventArgs e) {
    string databaseFile = AppDomain.CurrentDomain.BaseDirectory + txtDatenbankname.Text;
    File.Delete(databaseFile);
    SQLiteConnection.CreateFile(databaseFile);
    if (File.Exists(databaseFile)) {
        MessageBox.Show("Leere Datenbank \n\n'" + databaseFile + "'\n\n wurde erstellt.");
    }
}
```

Die Methode trägt den Pfad der **.exe**-Datei plus den vom Benutzer gewählten Datenbanknamen in die Variable **databaseFile** ein. Diese Datei wird, sofern vorhanden, gelöscht und dann mit der Methode **CreateFile** des Objekts **SQLiteConnection** neu erstellt. Dabei übergeben Sie die Variable mit dem Datenbankpfad als Parameter. Die folgende **if**-Bedingung prüft, ob nach

diesem Vorgang eine Datei mit dem angegebenen Namen vorhanden ist und zeigt in diesem Fall eine entsprechende Meldung an.

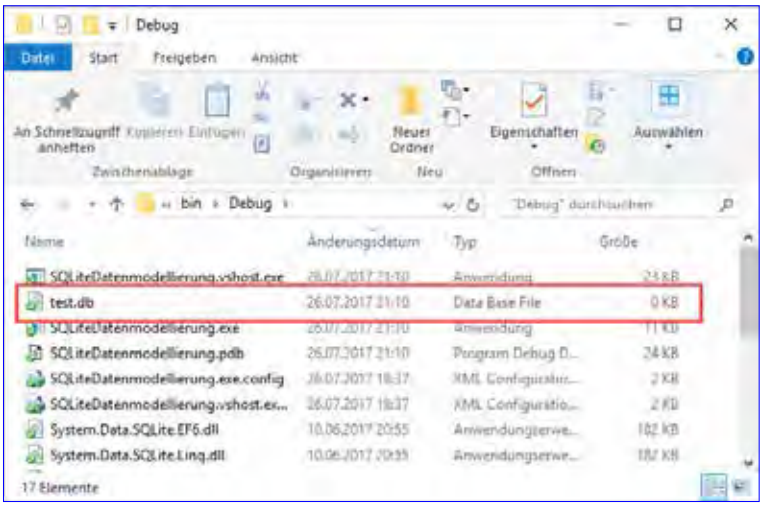


Bild 1: Frisch angelegte Datenbankdatei

Neue Tabelle erstellen

Nun wollen wir der frisch angelegten Datenbank eine erste Tabelle hinzufügen. Das ist dann auch tatsächlich der erste Inhalt der Datenbankdatei – diese weist nach dem Anlegen nämlich die Dateigröße **0** auf (siehe Bild 1). Danach rufen wir mit der Tasten **btnCreateTable** die folgende Methode auf, die wieder den Pfad der **.exe**-Datei und den Dateinamen aus dem Textfeld in der Variablen **databaseFile** zusammensetzt. Dann erstellt sie ein neues Objekt des Typs **SQLiteConnection**, der sie als Konstruktor-Parameter eine Zeichenkette übergibt, die aus **Data Source=**, dem Datenbankpfad und der zu verwenden Version besteht.

Die **Open**-Methode öffnet die Verbindung zu dieser Datenbank. Dann schreibt die Methode eine **CREATE TABLE**-Anweisung, die eine neue Tabelle namens **Beispieltabelle** mit den beiden Feldern **id** und **Beispielstring** enthält. Nun erstellt die Methode ein neues **SQLiteCommand**-Objekt und übergibt die SQL-Anweisung aus der String-Variablen **sql** sowie das Verbindungsobjekt aus **connection** als Parameter. Die **ExecuteNonQuery**-Methode führt die **CREATE TABLE**-Anweisung schließlich aus. Mit der **Close**-Methode schließen wir die Verbindung:

```
private void btnCreateTable_Click(object sender, RoutedEventArgs e) {
    string databaseFile = AppDomain.CurrentDomain.BaseDirectory + txtDatenbankname.Text;
    SQLiteConnection connection = new SQLiteConnection("Data Source=" + databaseFile + ";Version=3;");
    connection.Open();
    string sql = "CREATE TABLE Beispieltabelle (id INT, Beispielstring VARCHAR(255))";
    SQLiteCommand command = new SQLiteCommand(sql, connection);
    command.ExecuteNonQuery();
    connection.Close();
}
```

Ein Blick in den Windows Explorer zeigt, dass dies nicht spurlos an der Datenbankdatei vorübergegangen ist: Diese hat nun immerhin eine Größe von acht Kilobytes. Aber befindet sich auch die neue Tabelle darin? Dies prüfen wir mit dem Tool **SQLiteStudio**, das Sie kostenlos über das Internet herunterladen können. Mit dem Menüeintrag **Datenbank|Datenbank hinzufügen** öffnen Sie den Dialog aus Bild 2. Hier wählen Sie die soeben angelegte Datei aus und klicken nach dem Betätigen von **Verbindung testen** auf die Schaltlfäche **OK**.

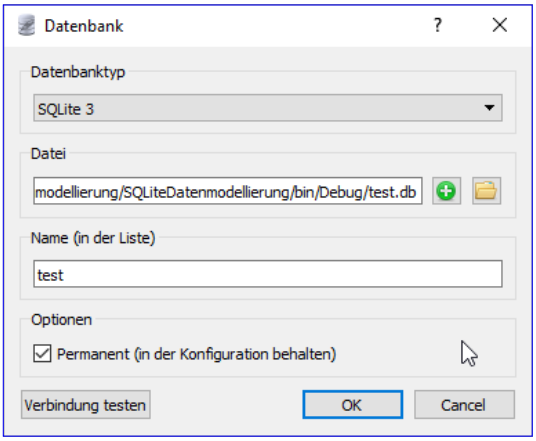


Bild 2: Hinzufügen der Datenbank zum **SQLiteStudio**

Danach finden Sie im Hauptfenster von **SQLiteStudio** bereits die Datenbank mit der ersten Tabelle im Datenbank-Explorer vor (siehe Bild 3). Allerdings verfügt das Feld **id** noch nicht über eine Primärschlüssel, aber das können wir ja gleich nachholen – indem wir die neue Tabelle verändern.

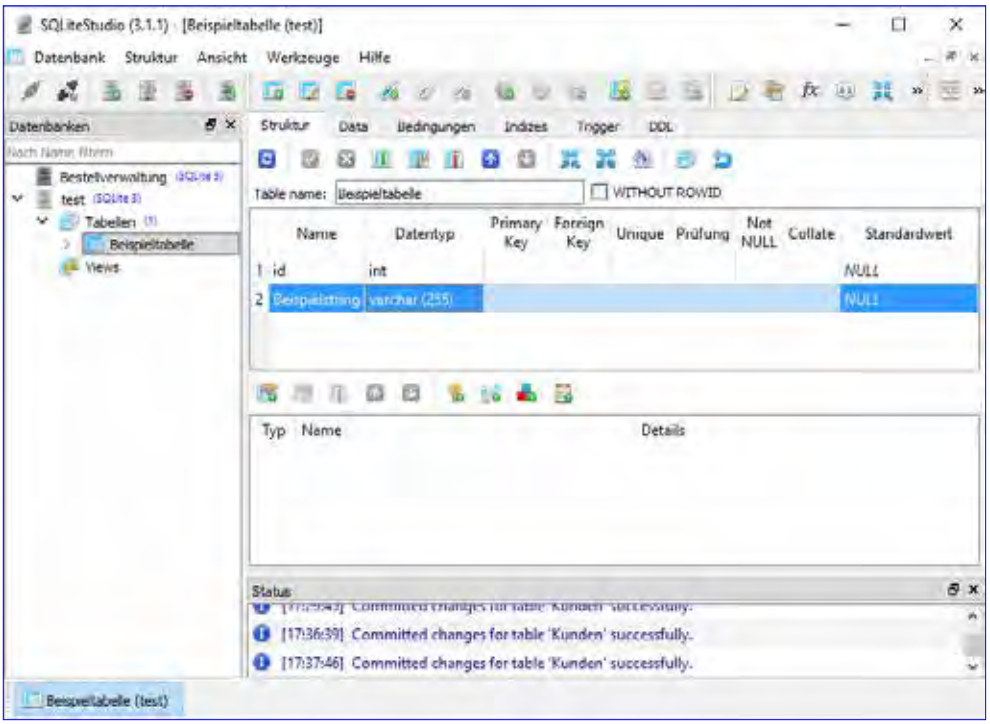


Bild 3: Die neue Tabelle im **SQLiteStudio**

Tabelle mit Primärschlüsselfeld erstellen

Im nächsten Beispiel wollen wir eine Tabelle erstellen und das **ID**-Feld direkt als Primärschlüsselfeld kennzeichnen. Dazu fügen wir der **CREATE TABLE**-Methode gleich das Schlüsselwort **PRIMARY KEY** für das Feld **ID** hinzu. Dazu nutzen wir prinzipiell die gleiche Methode wie für das vorherige Beispiel:

```
private void btnTabelleMitPKAnlegen_Click(object sender, RoutedEventArgs e) {
    string databaseFile = AppDomain.CurrentDomain.BaseDirectory + txtDatenbankname.Text;
    string connectionString = "Data Source=" + databaseFile + ";Version=3;";
    MessageBox.Show(connectionString);
    SQLiteConnection connection = new SQLiteConnection(connectionString);
    connection.Open();
    string sql = "CREATE TABLE BeispieltabelleMitPK (id INT PRIMARY KEY, Beispielstring VARCHAR(255))";
    SQLiteCommand command = new SQLiteCommand(sql, connection);
    command = new SQLiteCommand(sql, connection);
    command.ExecuteNonQuery();
    connection.Close();
}
```

Auch das liefert das gewünschte Ergebnis, wie ein Blick in **SQLiteStudio** beweist (siehe Bild 4).

Nun habe ich allerdings beim Erstellen der neuen Tabelle bei Experimentieren vergessen, das Textfeld **txtDatenbankname** zu füllen. Es gibt allerdings beim Durchlaufen der obigen Methode keinerlei Fehlermeldung – wie kann das sein? Theoretisch dürfte der Zugriff auf die Datenbank mit der **Open**-Methode doch gar nicht funktionieren, da die Datenbankdatei gar nicht vor-

handen ist? Doch das ist ein Fehlschluss: Dadurch, dass wir **txtDatenbankname** nicht gefüllt haben, übergeben wir nämlich einen String wie den folgenden mit **connectionString** an die **Open**-Methode:

```
Data Source=C:\Users\User\...\bin\
Debug\;Version=3;
```

Wenn wir nun in das Verzeichnis **...\bin\Debug** schauen, finden wir dort auch keine neue Datenbank etwa mit einem beim Öffnen temporär vergebenen Namen. Wenn wir allerdings eine Ebene nach oben gehen, also in das Verzeichnis **...\bin**, finden wir die Situation aus Bild 5 vor. SQLite legt, wenn die in der Verbindungszeichenfolge für die **Open**-Methode angegebene Datenbankdatei noch nicht vorhanden ist, eine neue Datenbank an. In diesem Fall hat SQLite **Debug\;Version=3;** als Dateinamen interpretiert und eine Datei namens **Debug;Version=3** angelegt (das Backslash-Zeichen **** wurde dabei als Escape-Zeichen für das Semikolon interpretiert). Die Größe von **20 KB** zeigt an, dass hier auch gleich die gewünschte Tabelle angelegt wurde.

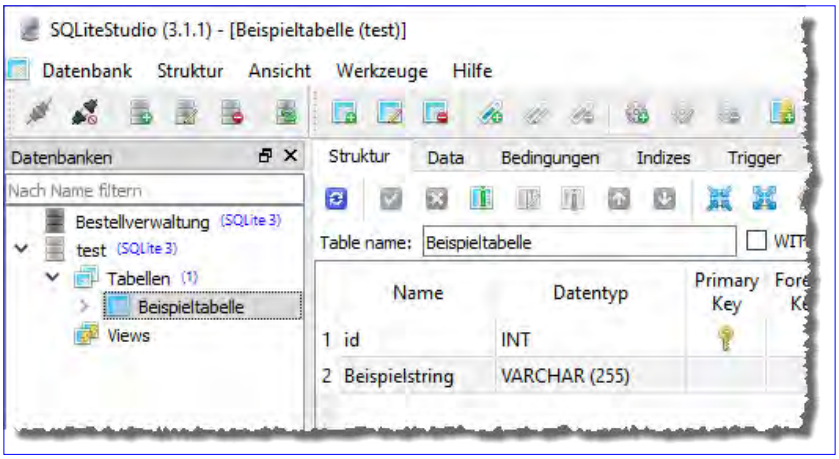


Bild 4: Tabelle mit Primärschlüssel im **SQLiteStudio**

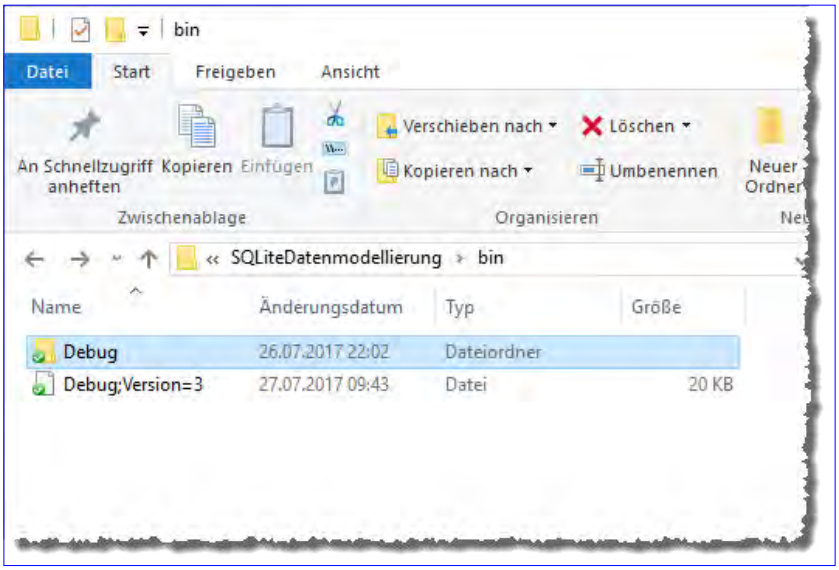


Bild 5: Eine Datenbankdatei namens **Debug;Version=3**

Autowert definieren

Beim Einfügen von Datensätzen müssen wir uns Gedanken darum machen, auf welche Weise der Wert des Primärschlüsselfeldes definiert wird. Unter Access wurde das immer einfach durch Festlegen der Eigenschaft **Autowert** für das betroffene Feld erledigt. Unter SQLite ist das auch nicht viel anders: Hier fügen Sie hinter dem **PRIMARY KEY**-Schlüsselwort einfach noch **AUTOINCREMENT** ein (siehe Methode **btnTabelleMitAutowertAnlegen_Click**):

```
CREATE TABLE BeispieltabelleMitAutowert(ID INTEGER PRIMARY KEY AUTOINCREMENT, ...)
```

SQLite hat hier allerdings eine etwas andere Philosophie als andere Datenbanksysteme: Sie müssen gar nicht unbedingt eine Autoincrement-Funktion definieren, sondern können das Primärschlüsselfeld auch einfach nur als **INTEGER PRIMARY KEY** definieren. Wenn Sie dann beim Neuanlegen eines Datensatzes keinen Wert für das Primärschlüsselfeld angeben, fügt SQLite automatisch einen entsprechend ermittelten Wert ein. Gegenüber anderen Datenbanksystemen haben Sie so die Wahl, ob Sie

das System einen Primärschlüsselwert auswählen lassen wollen oder ob Sie diesen selbst vergeben wollen. Mehr dazu erfahren Sie weiter unten unter **Datensätze einfügen**.

Prüfen, ob Datenbank vorhanden ist

Wir wollen an dieser Stelle den Benutzer darauf hinweisen, dass die Datenbank noch nicht vorhanden ist und die Methode abbrechen. Das erledigen wir mit einer entsprechenden **if**-Bedingung:

```
private void btnTabelleMitPKAnlegen_Click(object sender, RoutedEventArgs e) {
    string databaseFile = AppDomain.CurrentDomain.BaseDirectory + txtDatenbankname.Text;
    if (!File.Exists(databaseFile)) {
        MessageBox.Show("Die Datenbank \n\n'" + databaseFile + "'\n\nist nicht vorhanden.");
    }
    else {
       //Tabelle anlegen
    }
}
```

Herstellen einer Verbindung in eigener Methode

In den bisherigen Beispielen haben wir jeweils einige Anweisungen eingebaut, welche die Verbindung herstellen und mit der Variablen **connection** referenzieren. Damit wir diese Zeilen nicht in jeder neuen Methode erneut schreiben müssen, gliedern wir diese in eine eigene Methode aus und erweitern deren Funktionsumfang gleich ein wenig.

Die neue Methode heißt VerbindungOeffnen und erwartet den Pfad zur Datenbankdatei als Parameter (**datenbankpfad**) sowie einen Boolean-Wert namens **datenbankErstellen**, der angibt, ob die Datenbank erstellt werden soll, wenn Sie noch nicht vorhanden ist. Die Methode stellt die Boolean-Variable **datenbankExistiert** zunächst auf **true** ein. Dann prüft sie, ob die mit **datenbankpfad** angegebene Datei existiert. Falls nicht, wird **datenbankExistiert** auf **false** eingestellt und eine Meldung ausgegeben, dass die Datenbank nicht vorhanden ist. Falls die Datenbank existiert, baut die Methode in der zweiten **if**-Bedingung eine Verbindungszeichenfolge zusammen, erstellt ein **SQLiteConnectionString**-Objekt auf Basis dieser Verbindung und öffnet diese mit der **Open**-Methode. Die Verbindung wird dann als Rückgabewert zurückgeliefert. Sollte **datenbankExistiert** den Wert **false** enthalten, liefert die Methode den Wert **null** zurück:

```
private SQLiteConnection VerbindungOeffnen(string datenbankpfad, bool datenbankErstellen = false) {
    bool datenbankExistiert = true;
    if (!File.Exists(datenbankpfad)) {
        if (!datenbankErstellen) {
            datenbankExistiert = false;
            MessageBox.Show("Die Datenbank \n\n'" + datenbankpfad+ "'\n\nist nicht vorhanden.");
        }
    }
    if (datenbankExistiert) {
        string connectionString = "Data Source=" + datenbankpfad + ";Version=3;";
```

```
        SQLiteConnection connection = new SQLiteConnection(connectionString);
        connection.Open();
        return connection;
    }
    else {
        return null;
    }
}
```

Prüfen, ob Tabelle vorhanden ist

Der nächste Schritt beim Anlegen einer Tabelle ist eine vorherige Prüfung, ob diese Tabelle bereits vorhanden ist. Wenn wir eine Connection haben und den Namen der zu prüfenden Tabelle, können Sie die folgende Methode verwenden, um zu prüfen, ob die Tabelle bereits vorhanden ist. Die Methode erwartet die Verbindung (**connection**) und den Tabellennamen (**tabellenname**) als Parameter. Sie fügt in sql eine SQL-Abfrage zusammen, welche die Tabelle **sqlite_master** nach einem Datensatz durchsucht, dessen Feld **type** den Wert **table** und dessen Feld **name** den übergebenen Tabellennamen enthält. Diese Abfrage übergeben wir wieder mit der Connection an ein neues **SQLiteCommand**-Objekt. Wir führen dieses allerdings diesmal nicht mit der Methode **ExecuteNonQuery** aus, sondern mit **ExecuteScalar**. Dies liefert die erste Spalte der ersten Zeile des Ergebnisses zurück. Wir haben für die erste Spalte den Wert **1** angegeben. Findet **ExecuteScalar** keinen passenden Datensatz, ist die Tabelle nicht vorhanden und es wird der Wert **null** zurückgeliefert. Als Rückgabewert nutzen wir einen Boolean-Wert, der angibt, ob die Variable **tabelleVorhanden**, die nun entweder den Wert **1** oder **null** enthält, nicht den Wert **null** aufweist – also den Wert **true**, wenn die Tabelle gefunden wurde und **false**, falls nicht:

```
private bool TabelleVorhanden(SQLiteConnection connection, string tabellenname) {
    string sql = "SELECT 1 FROM sqlite_master WHERE type = 'table' AND name = '" + tabellenname + "'";
    SQLiteCommand command = new SQLiteCommand(sql, connection);
    var tabelleVorhanden = command.ExecuteScalar();
    return (tabelleVorhanden != null);
}
```

Tabelle nur anlegen, wenn noch nicht vorhanden

Mit den beiden zuvor erstellten Methoden können wir den Code für das Anlegen einer Tabelle nur in dem Fall, dass diese noch nicht vorhanden ist, reduzieren. Dies sieht dann wie folgt aus:

```
private void btnPruefenObTabelleVorhanden_Click(object sender, RoutedEventArgs e) {
    string databaseFile = AppDomain.CurrentDomain.BaseDirectory + txtDatenbankname.Text;
    SQLiteConnection connection = VerbindungOeffnen(databaseFile, false);
    if (connection != null) {
        bool tabelleVorhanden = TabelleVorhanden(connection, "BeispieltabelleMitPK");
        if (!tabelleVorhanden) {
            string sql = "CREATE TABLE BeispieltabelleMitPK (id INT PRIMARY KEY, Beispielstring VARCHAR(255))";
            SQLiteCommand command = new SQLiteCommand(sql, connection);
```

Entity Framework: SQLite verknüpfen

In den letzten Artikel haben wir oft die SQLite-Datenbank als Beispieldatenbank verwendet. Nun kann es allerdings durch einen falschen Connection-String oder eine verschobene, gelöschte oder leere Datenbank dazu kommen, dass die Anwendung nicht über das Entity Framework auf die Datenbank zugreifen kann. Dieser Artikel beschreibt einige Szenarien, in denen das der Fall ist und zeigt, wie Sie beim Start der Anwendung prüfen können, ob die Quell-Datenbank erreichbar ist und diese gegebenenfalls neu auswählen.

Die SQLite-Datenbank ist eine sehr praktische Datenquelle für einfache Anwendungen, die keinen SQL Server benötigen. Sie kommt als eine einzige Datei daher und die Treiber können bequem in ein Projekt integriert werden – auch die Weitergabe ist überhaupt kein Problem. Was aber ein Problem sein könnte, wissen Access-Entwickler, die Datenbanken in Front- und Backend aufgeteilt haben, nur zu gut: Die Datenbankdatei könnte sich nicht an dem Ort befinden, der in der Verbindungszeichenfolge des Frontends angegeben ist.

Mit einem .NET-WPF-Projekt und einer SQLite-Datenbank haben Sie ein klassisches Frontend-Backend-Szenario, wie es auch bei aufgeteilten Access-Datenbanken vorliegt. Im Gegensatz zur Konstellation mit einem SQL Server haben Sie keine Server-Angabe im Connection-String, sondern Sie müssen den Namen der Datenbank, gegebenenfalls mit Angabe des Pfades zu dieser Datenbank, angeben. Wenn Sie ein Entity Data Model zu Ihrem Projekt hinzugefügt und die Datenbankdatei, zum Beispiel **Bestellverwaltung.db**, als Datenquelle ausgewählt haben, schreibt der Entity Data Model-Assistent eine Verbindungszeichenfolge in die Datei **App.config**, die etwa wie folgt aussieht:

```
<configuration>
  ...
  <connectionStrings>
    <add name="BestellverwaltungEntities" connectionString="...;provider=System.Data.SQLite.EF6;provider connection
        string=&amp;quot;data source=c:\Beispiel\Bestellverwaltung.db&amp;quot;" providerName="System.Data.EntityClient" />
  </connectionStrings>
</configuration>
```

Hier sehen Sie, dass der Assistent zum Erstellen des Entity Data Models den absoluten Pfad zur Datenbankdatei **Bestellverwaltung.db** angegeben hat. Das geht solange gut, wie Sie die Anwendung auf dem Entwicklungsrechner debuggen oder einsetzen und die Datenbank sich an Ort und Stelle befindet.

Wenn Sie jedoch die Anwendung auf einen anderen Rechner verschieben, ist die Wahrscheinlichkeit hoch, dass die Datenbankdatei unter diesem Pfad nicht mehr zu erreichen ist. Ich selbst programmiere beispielsweise auf zwei verschiedenen Rechnern, wobei die Dateien per DropBox jeweils auf den aktuellsten Stand gebracht werden. Und sobald das Projekt samt Datenbankdatei auf dem einen Rechner nicht auf dem gleichen Pfad wie auf dem anderen Rechner liegt, findet eines der beiden Projekte die Datenbankdatei nicht mehr.

Relativ statt absolut

In diesem Fall ist die erste Maßnahme, den Pfad aus der Angabe der Quelldatenbank zu entfernen und nur noch den Dateinamen anzugeben. Das sieht dann in der Datei **App.config** wie folgt aus:

```
<configuration>
  ...
  <connectionStrings>
    <add name="BestellverwaltungEntities" connectionString="...;provider=System.Data.SQLite.EF6;provider connection
        string=&quot;data source=Bestellverwaltung.db&quot;" providerName="System.Data.EntityClient" />
  </connectionStrings>
</configuration>
```

Dies bedeutet nun nichts anderes, als dass die Datenbankdatei nun immer dann erreichbar ist, wenn sich diese im gleichen Verzeichnis wie die **.exe**-Datei des Projekts befindet.

Mehrbenutzer-Szenario

Das wird in einem Mehrbenutzer-Szenario natürlich nicht mehr funktionieren, denn schon wenn die Anwendung auf zwei Rechnern installiert ist, müsste man auf einem Rechner den absoluten Pfad zu der zu nutzenden Datenbankdatei hinterlegen.

Wir benötigen also einen Mechanismus, der beim Start der Anwendung ausgelöst wird und prüft, ob die zu verwendende Datenbankdatei verfügbar ist. Wenn dies nicht der Fall ist, soll ein Dialog erscheinen, mit dem der Benutzer die Datenbankdatei auswählen soll. Diese Änderung sollte dann optimalerweise irgendwo in den Einstellungen der Datenbank gespeichert und von nun an beim Start abgerufen werden können. Wir beginnen jedoch mit der Prüfung, ob die Datenbankdatei überhaupt an Ort und Stelle ist. Dies erledigen wir beim Start der Anwendung, und zwar beim Aufruf des Hauptfensters.

Beispielkonstellation

Wir erstellen im Fenster MainWindow eine minimale Konfiguration, um die Kunden aus der Datenbank **Bestellverwaltung.db** in einem DataGrid anzuzeigen. Dazu fügen Sie **MainWindow.xaml** folgenden Code hinzu:

```
<Grid>
    <DataGrid ItemsSource="{Binding Kunden}"></DataGrid>
</Grid>
```

Das Code behind-Modul MainWindow.xaml.cs statten wir so aus:

```
using System.Windows;
using System.Collections.ObjectModel;
namespace TestDatenbankVorhanden {
    public partial class MainWindow : Window {
        BestellverwaltungEntities dbContext = null;
        private ObservableCollection<Kunde> kunden;
```

```
        public ObservableCollection<Kunde> Kunden {
            get { return kunden; }
            set { kunden = value; }
        }
        public MainWindow() {
            InitializeComponent();
            dbContext = new BestellverwaltungEntities();
            kunden = new ObservableCollection<Kunde>(dbContext.Kunden);
            DataContext = this;
        }
    }
}
```

Damit werden die Daten im Fenster angezeigt.

Datenquelle entfernen

Was geschieht nun, wenn wir die Datenquelle vom gewohnten Ort entfernen? Dazu wechseln Sie im Windows Explorer zum Verzeichnis **bin\Debug** unterhalb des Projektordners und benennen die Datei **Bestellverwaltung.db** um, zum Beispiel in **_Bestellverwaltung.db**. Starten Sie die Anwendung nun von diesem Verzeichnis aus per Doppelklick auf den Namen der **.exe**-Datei (im Falle der Beispielanwendung **TestDatenbankVorhanden.exe**). Werden die Daten nun immer noch angezeigt? Dann stehen die Chancen gut, dass Sie im Connection-String noch den absoluten Pfad zur Datei **Bestellverwaltung.db** stehen haben und die **.exe**-Datei auf diese Datei zugreift. Entfernen Sie hier zunächst, wie oben beschrieben, die Pfadangabe, sodass nur noch der Dateiname **Bestellverwaltung.db** im Connection-String steht. Starten Sie das Projekt dann neu, damit die Anwendung mit der neuen Verbindungszeichenfolge im Verzeichnis **bin/Debug** landet. Dadurch wird nun auch wieder die Datei **Bestellverwaltung.db** eingetragen, deren Namen Sie ja soeben schon in **_Bestellverwaltung.db** geändert haben. Beenden Sie die Anwendung also wieder und benennen Sie die Datei **Bestellverwaltung.db** erneut in **_Bestellverwaltung.db** um. Starten Sie dann die Anwendung **TestDatenbankVorhanden.exe** erneut. Was geschieht nun? Normalerweise sollte nun nichts Sichtbares geschehen – die Anwendung wird einfach nicht gestartet. Das liegt daran, dass wir keine Fehlerbehandlung eingebaut haben, die Anwendung aber beim Aufrufen einen Fehler auslöst, denn die Datenquelle ist ja nicht verfügbar. Interessanterweise finden Sie im Verzeichnis **bin/Debug** aber nun eine neuen Datei namens **Bestellverwaltung.db** mit einer Größe von **0 KB** vor. Dies geschieht, weil das Entity Framework eine neue Datenbankdatei erstellt, wenn beim Versuch des Zugriffs keine entsprechende Datenbankdatei gefunden werden konnte.

Fehler per Exception-Handling aufdecken

Wir wollen die Anwendung also nun mit einer Fehlerbehandlung erweitern, damit wir die Fehlermeldung erhalten und entsprechend darauf reagieren können. Unsere Vermutung ist, dass der Fehler beim Zugriff auf die **Kunden**-Liste auftritt. Die betroffene Anweisung fassen wir also in einen **try**-Block ein. Zusammen mit dem **catch**-Block sieht das Ganze nun wie folgt aus:

```
try {
    kunden = new ObservableCollection<Kunde>(dbContext.Kunden);
}
```

```
catch (Exception e) {
    MessageBox.Show("Error:\n\n" + e.Message + "\n\n" + e.InnerException.Message);
}
```

Wir erhalten dann die folgende Fehlermeldung:

```
Error:
An error occurred while executing the command definition. See the inner exception for details.
SQL logic error or missing database
no such table: Kunden
```

Damit können wir etwas anfangen, denn wir wissen ja bereits, dass die Datenbank nicht vorhanden ist. Aber können wir dies auch prüfen, ohne einen Fehler auszulösen? Beispielsweise, indem wir einfach testen, ob die gesuchte Datenbankdatei an der angegebenen Stelle ist? Nein, denn erstens wissen wir ja aktuell gar nicht, wo sich diese befinden soll – dazu müssten wir erst die Verbindungszeichenfolge auslesen. Danach würden wir dann prüfen, ob die entsprechende Datei vorhanden ist. Allerdings kann es auch sein, dass diese Datei beim vorherigen Zugriffsversuch angelegt wurde, aber keine Daten enthält. In diesem Fall würden wir eine Datei vorfinden, erhielten aber einen Fehler, weil die Tabellen, auf die wir zugreifen wollen, nicht mehr vorhanden sind. Der Versuch, auf die Daten einer Tabelle der Datenbank zuzugreifen und diesen Zugriff in eine Ausnahmebehandlung zu integrieren, scheint also durchaus die einfachste Wahl zu sein.

Passende Datenbank auswählen

Sollte der Zugriff auf eine der Tabellen der Datenbank nicht gelingen, soll der Benutzer die Möglichkeit erhalten, die Datenbankdatei auszuwählen. Der Pfad zur Datenbank soll dann in der Konfiguration der Anwendung gespeichert und beim nächsten Öffnen wieder abgerufen werden.

Dazu legen wir zunächst eine geeigenete Einstellung für die Anwendung an. Klicken Sie doppelt auf den Eintrag **Properties** im Projektmappen-Explorer. Im nun erscheinenden Fenster wählen Sie links den Bereich Einstellungen aus. Hier tragen Sie nun für den ersten Eintrag der Liste den Wert **DatabasePath** unter **Name** ein, die übrigen Werte können Sie wie in Bild 1 beibehalten. Speichern und schließen Sie diesen Bereich dann.

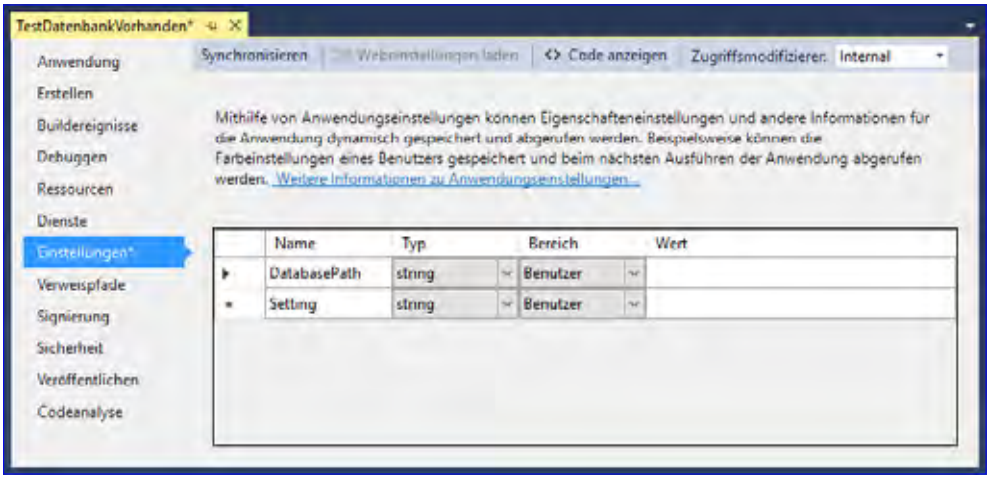


Bild 1: Definition einer Einstellung zum Speichern des Datenbankpfades

Pfad testen und abfragen

Wie wir testen, ob die Verbindungszeichenfolge auf eine gültige Datenbankdatei verweist, haben wir ja weiter

Weitergabe von WPF/C#/SQLite-Anwendungen

Die bisher bekannte Methode im Rahmen der Entwicklung von Anwendung in diesem Magazin war das Kopieren des bin/debug-Verzeichnisses auf den Zielrechner. In dieser Ausgabe wollen wir uns nun ansehen, wie Sie Anwendungen mit den Bordmitteln von Visual Studio 2015 in der Community-Edition weitergeben können. Die Möglichkeiten sind beschränkt, aber für die meisten Fälle durchaus ausreichend.

Voraussetzungen

Für Testzwecke wollen wir unser bisher in diesem Magazin besprochene Anwendung **Bestellverwaltung** weitergeben. Dazu benötigen Sie die Erweiterung **Visual Studio Installer Projects**, die Sie über den Dialog **Extensions und Updates** installieren können (Menüeintrag **Extras|Extensions und Updates...**). Klicken Sie dort auf **Online** und geben Sie **Install** als Suchbegriff ein. Sie finden dann den Eintrag **Microsoft Visual Studio 2015 Installer Projects**, den Sie per Mausklick herunterladen und anschließend installieren können.

Weitergabe-Projekte

Visual Studio bietet, wenn Sie ein neues Projekt erstellen, einige Einträge im Bereich **Vorlagen|Andere Projekttypen|Visual Studio Installer** (siehe Bild 1).

Von diesen Projekttypen ist für unsere Zwecke nur einer interessant, nämlich **Setup Project**: Dieser erstellt eine Datei mit dem Namen **Setup.exe** sowie eine weitere namens **<Projektname>.msi**.

Aus zwei mach eins

Nun werden Sie sich vermutlich fragen, wie Sie die Anwendung auf Basis des weiterzugebenden Projekts und das Setup-Projekt zusammenbringen sollen. Das ist gar nicht so schwer: Wir öffnen einfach die Projektmappe mit dem weiterzugebenden Projekt (also die **.sln**-Datei) und fügen der Projektmappe das Setup-Projekt hinzu. Dazu klicken Sie im Projektmappen-Explorer mit der rechten Maustaste auf das Projektmappen-Element und wählen aus dem

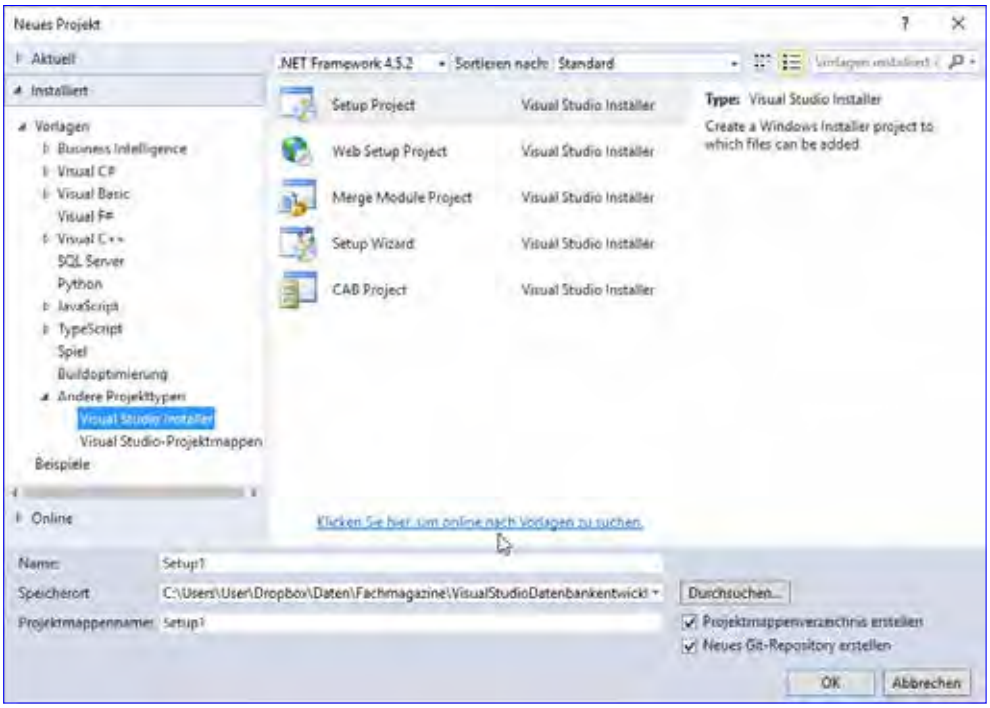


Bild 1: Weitergabe-Projekte

Kontextmenü den Eintrag **Hinzufügen|Neues Projekt** aus (siehe Bild 2). Im nun erscheinenden Dialog **Neues Projekt** wählen Sie dann die bereits weiter oben genannte Projektvorlage **Setup Project** aus und erstellen diese unter dem Namen **InstallBestellverwaltung** im Verzeichnis der Projektmappe, welches hier bereits voreingestellt sein dürfte (diese Bezeichnung wird als Name für die **.msi**-Datei verwendet und im Setup-Dialog als Titelzeile eingeblendet, also wählen Sie diesen sorgfältig aus). Im Gegensatz zu dem oben abgebildeten Dialog **Neues Projekt** enthält der vom Projektmappen-Explorer aus gestartete Dialog nicht das Textfeld **Projektmappenname**. Logisch, denn wir fügen ja einer Projektmappe ein neues Projekt hinzu.

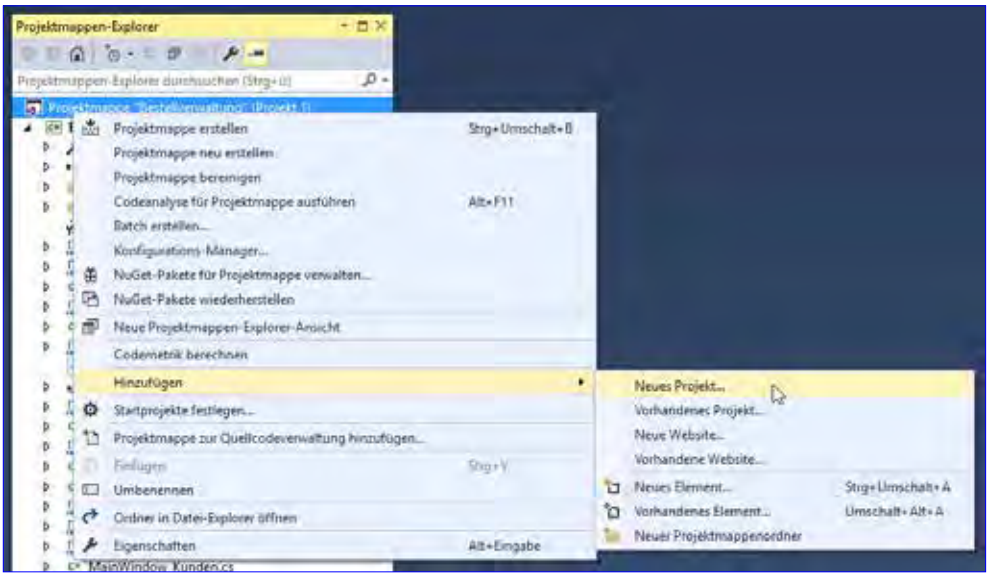


Bild 2: Hinzufügen eines Setup-Projekts

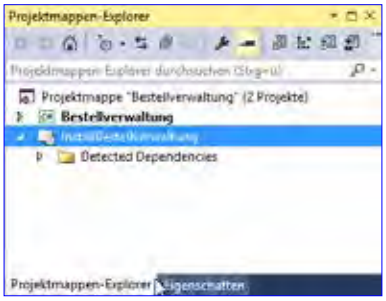


Bild 3: Zwei Projekte in einer Projektmappe

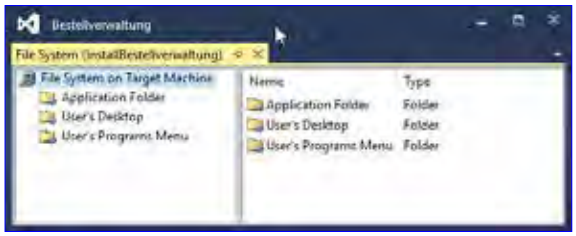


Bild 4: Der nach dem Erstellen des Setup-Projekts eingeblendete **FileSystem**-Editor.

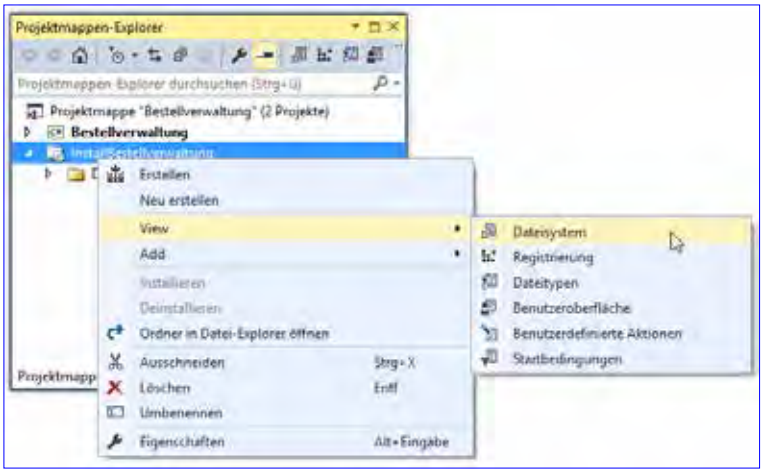


Bild 5: Aufruf der verschiedenen Editoren des Setups

Nach dem Erstellen des neuen Projekts erscheint dieses im Projektmappen-Explorer in der gleichen Ebene wie das Projekt selbst (siehe Bild 3).

Dateisystem-Editor

Nun erscheint der **FileSystem**-Editor (siehe Bild 4).

Sollten Sie diesen einmal versehentlich schließen, können Sie ihn über den Kontextmenü-Eintrag **View|Dateisystem** des Setup-Projekts wieder einblenden (siehe Bild 5). Dies gilt auch für die anderen Editoren des Setups, auf die wir später zu sprechen kommen.

Quelle hinzufügen

Nun fügen wir allerdings erst einmal die Information über die Dateien hinzu, die im Setup landen sollen. Dazu klicken Sie mit der rechten Maustaste auf den Eintrag **Application Folder** im Dateisystem-Editor und wählen dort den Kontextmenü-Befehl **AddlProjektausgabe** aus (siehe Bild 6).

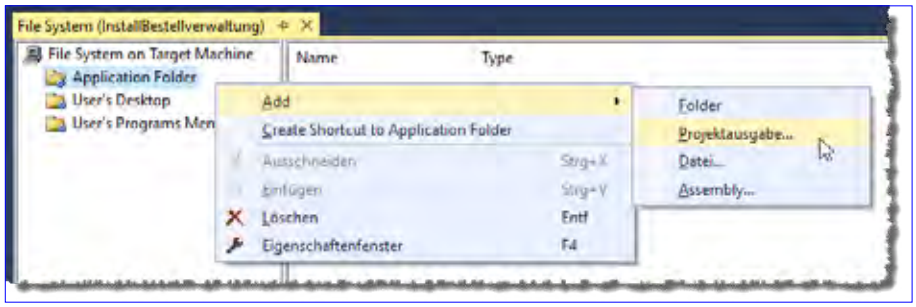


Bild 6: Hinzufügen der Projektausgabe

Damit öffnen Sie den Dialog aus Bild 7, der im oberen Kombinationsfeld den Namen des Projekts enthalten sollte, dessen Ausgabe in das Setup aufgenommen werden soll – hier also **Bestellverwaltung**. In der Liste darunter soll der Eintrag **Primäre Ausgabe** markiert sein. Klicken Sie auf **OK**, um den Dialog zu schließen.

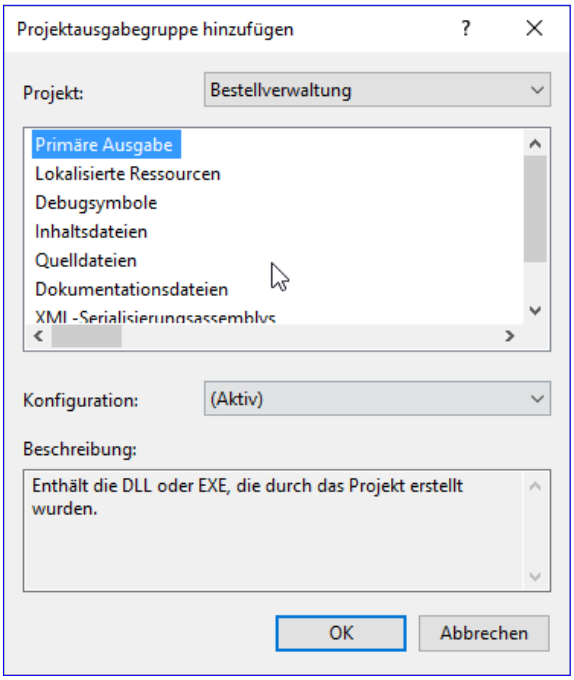


Bild 7: Projektausgabegruppe festlegen

Damit kommt Leben in die Sache. Sowohl im Projektmappen-Explorer, der nun einige Einträge unterhalb des bis dahin leeren Elements **Detected Dependencies** enthält, landen diese auch im Ordner **Application Folders** des **FileSystem**-Editors – nebst dem Element **Primäre Ausgabe from Bestellverwaltung (Active)** (siehe Bild 8).

Zielverzeichnis einstellen

Nun legen wir fest, wo auf dem Zielrechner die Anwendung landen soll. Dazu klicken Sie wiederum mit der rechten Maustaste auf den Eintrag **Application Folder** und wählen diesmal den Kontextmenü-

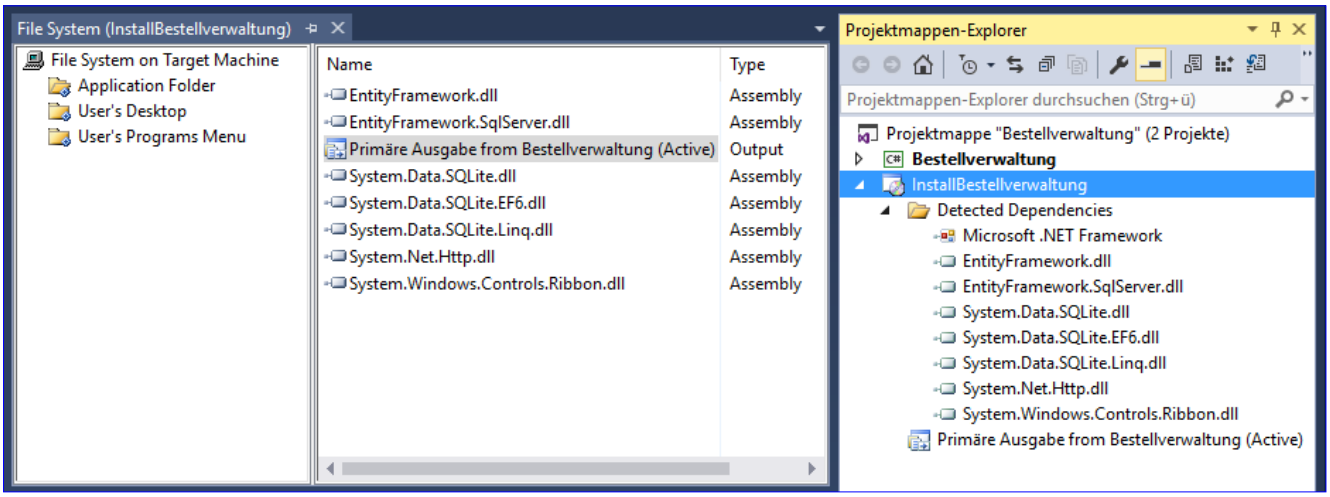


Bild 8: Gefüllter **Application Folder**

Eintrag **Eigenschaftenfenster** aus. Hier finden Sie die Eigenschaft **DefaultLocation** vor, die in unserem Beispiel standardmäßig den Wert **[ProgramFilesFolder][Manufacturer]\[ProductName]** enthält (siehe Bild 9). Die einzelnen Elemente werden bei der Installation ersetzt. Den Wert für **ProgramFilesFolder** ermittelt das Setup abhängig vom Zielsystem. Die beiden anderen Einstellungen, **Manufacturer** und **ProductName**, können Sie selbst in den Eigenschaften des Projekts festlegen.

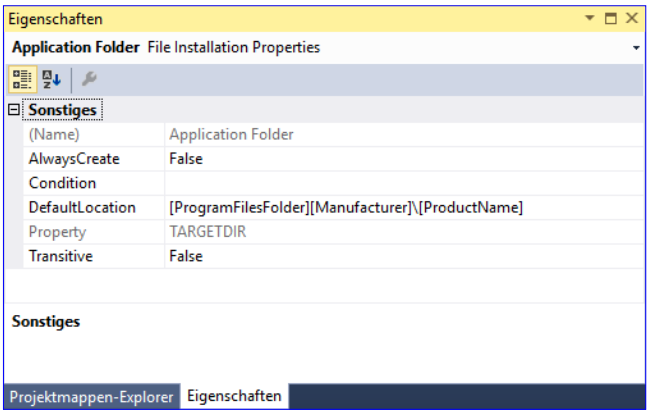


Bild 9: Zielverzeichnis festlegen

Dazu wechseln Sie im Projektmappen-Explorer zu dem Eintrag für das Install-Projekt (hier **InstallBestellverwaltung**). Aktivieren Sie mit **F4** die Anzeige der Eigenschaften, erscheint der Dialog aus Bild 10. Hier stellen Sie dann etwa die Eigenschaften **Manufacturer**, **ProductName** oder **Title** ein. Wir haben hier Werte verwendet, mit denen wir bei einem ersten Test schnell erkennen können, welche Eigenschaft wo im Setup landet – also etwa **Eigenschaften_Manufacturer** für **Manufacturer**.

Setup erstellen

Danach erstellen wir das Setup. Dazu speichern Sie vorsichtshalber alle Änderungen (Menüeintrag **Datei|Alles speichern**). Anschließend öffnen Sie mit dem Menüeintrag **Erstellen|Konfigurations-Manager...** den **Konfigurations-Manager**. Dieser zeigt die beiden Projekte der aktuellen Projektmappe an. Hier stellen Sie nun für beide Einträge unter **Konfiguration** den Wert **Release** ein und markieren jeweils die Option **Erstellen** (siehe Bild 11). Schließen Sie den Dialog dann wieder.

Wählen Sie dann den Menüeintrag **Erstellen|Projektmappe erstellen** aus. Nach dem Erstellen des Setups finden Sie im Ordner **...InstallBestellverwaltung\Release** die beiden Dateien **InstallBestellverwaltung.msi** und **setup.exe** (siehe Bild 12).

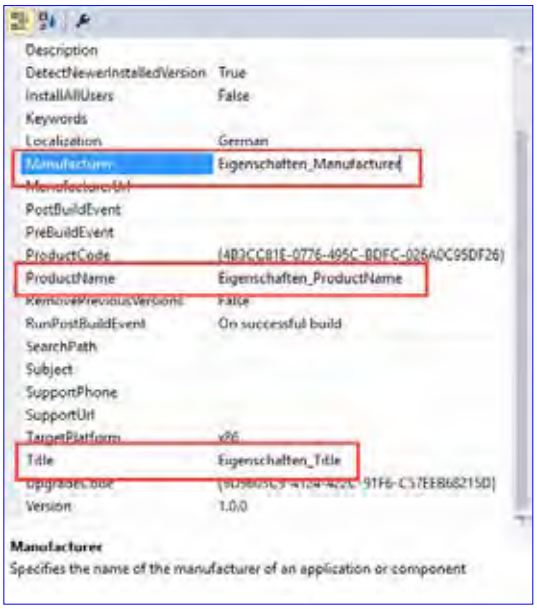


Bild 10: Einstellen der Setup-Eigenschaften

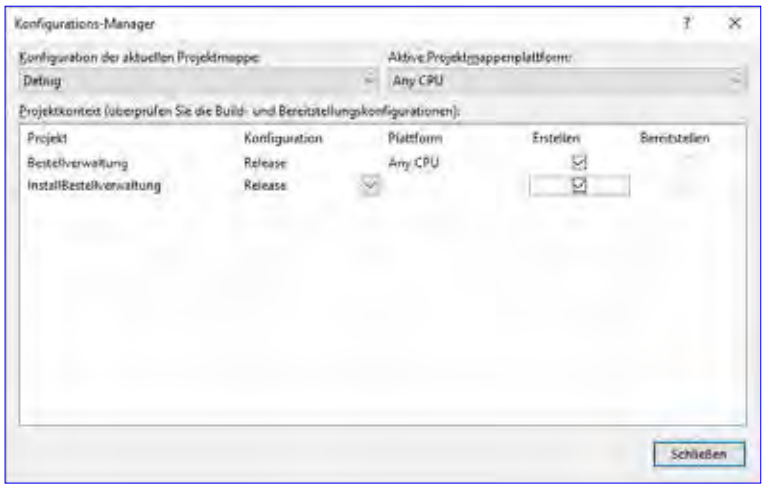


Bild 11: Konfigurieren der Erstellung

Unterschied .msi und setup.exe

Wozu benötigen Sie diese beiden Dateien? Aktuelle Windows-Systeme sind mit dem Windows Installer ausgestattet. Klicken Sie doppelt auf die **.msi**-Datei, wird diese mit dem Windows Installer geöffnet. Die **.msi**-Datei ist eine Datenbankdatei, welche Informationen über die zu installierende Anwendung und gegebenenfalls die Anwendung und weitere benötigte Dateien selbst enthält. Die Datei **setup.exe** ist eine Bootstrapper-Datei. Diese führt nicht die Installation durch, sondern prüft, ob die korrekte Version des Windows Installers auf dem Zielsystem installiert ist und ruft diese dann für die **.msi**-Datei auf.

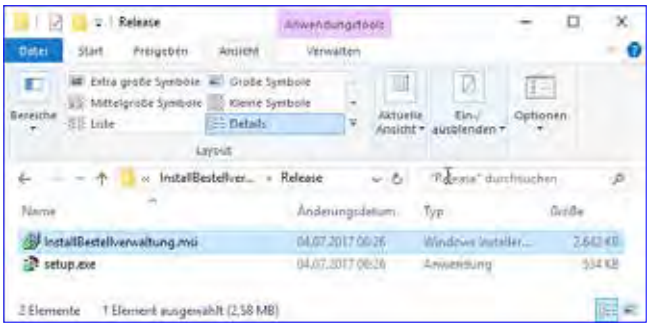


Bild 12: Setup-Dateien

Test der Installation

Nun schauen wir uns an, ob die Installation funktioniert und klicken dazu doppelt auf die **.msi**-Datei. Diese zeigt nun einen typischen Installationsdialog an (siehe Bild 13).

Nach diesem ersten allgemeinen Schritt folgt gleich der Schritt mit den meisten auszuwählenden Optionen (siehe Bild 14). Hier legen Sie fest, in welches Verzeichnis die Anwendung installiert werden soll, können den Speicherbedarf abfragen und festlegen, ob die Anwendung für alle oder nur für den aktuellen Benutzer installiert werden soll.

Schließlich starten Sie im dritten Dialog die Installation (siehe Bild 15). Im Anschluss erscheinen dann noch die Abfrage der

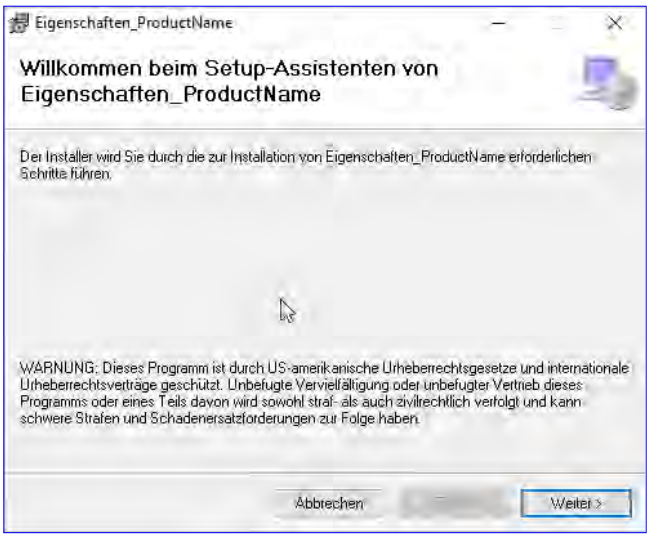
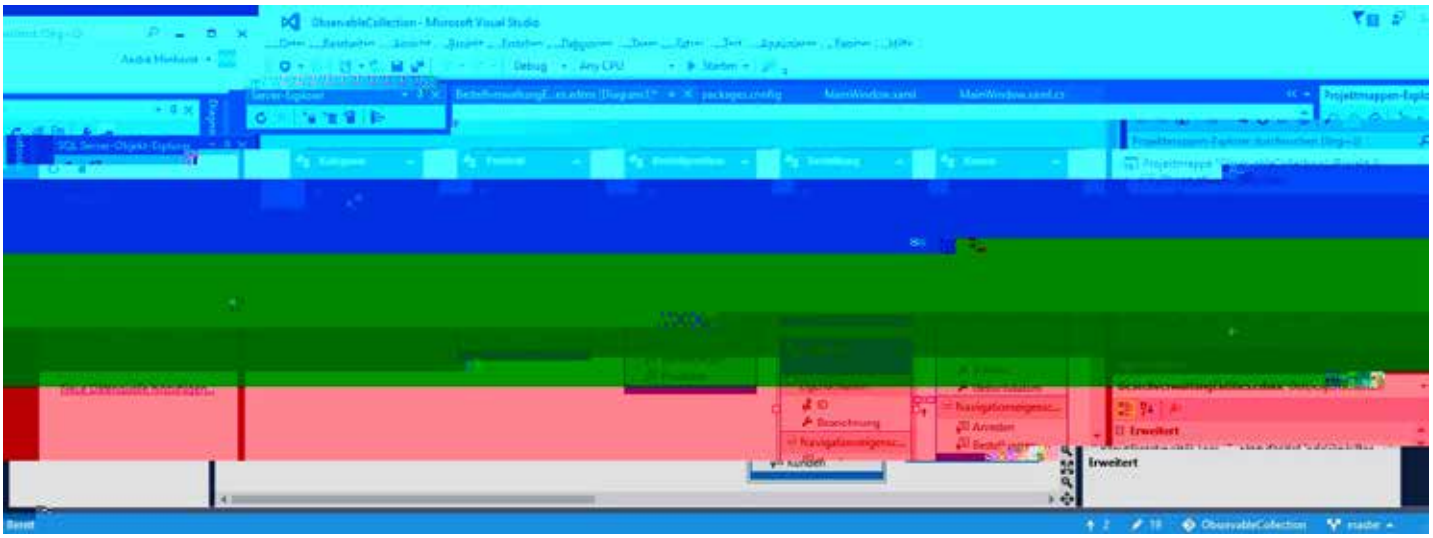


Bild 13: Erster Installationsschritt

AUSGABE 04/2017

DATENBANK ENTWICKLER

MAGAZIN FÜR DIE DATENBANKENTWICKLUNG MIT VISUAL STUDIO FÜR DESKTOP, WEB UND CO.



TOP-THEMEN:

C#-BASICS	Basics: ObservableCollection	SEITE 3
USERINTERFACE	Neuer Eintrag in ComboBox	SEITE 18
USERINTERFACE	m:n-Beziehung mit Listenfeld	SEITE 25
DATENZUGRIFF	LINQ to Entities: Methodensyntax	SEITE 42
ANWENDUNGEN	EDM: Backend ändern	SEITE 53

Basics: ObservableCollection

Unter WPF gibt es einige Mechanismen, welche die Bindung der Steuerelemente an die zugrunde liegenden Daten in einem gewissen Rahmen automatisieren. Diese werden durch die Programmierung bestimmt – entweder durch die Verwendung bestimmter Schnittstellen für Eigenschaften oder auch durch entsprechende Auflistungstypen, die dann als Datenquelle etwa für Listen-Elemente verwendet werden. Mit der PropertyChanged-Schnittstelle haben wir schon die Synchronisierung zwischen den Attributen der XAML-Definition und Eigenschaften in den Code-Klassen besprochen (siehe Artikel »Basics: PropertyChanged«). In diesem Artikel schauen wir uns nun den Auflistungstyp ObservableCollection an.

Beispielanwendung

Die Beispielanwendung verwendet die auch in den anderen Artikeln genutzte SQLite-Datenbank **Bestellverwaltung.db**. Wir wollen daraus die Tabelle **Kunden** nutzen, um die Unterschiede zwischen einer normalen Liste (**List**) und einer **ObservableCollection** zu demonstrieren.

Die **ObservableCollection**-Klasse ist eine Alternative beispielsweise zur **List**-Klasse. Der wesentliche Unterschied ist, dass die **ObservableCollection**-Klasse externe Objekte, die den Inhalt einer **ObservableCollection** anzeigen, über Änderungen informiert.

Änderungen können dabei etwa das Hinzufügen oder Entfernen von Elementen sein. Beispiele für Steuerelemente, die Sie an Objekte auf Basis der Klasse **ObservableCollection** binden können, sind etwa das **ListBox**-Element oder das **DataGrid**-Element.

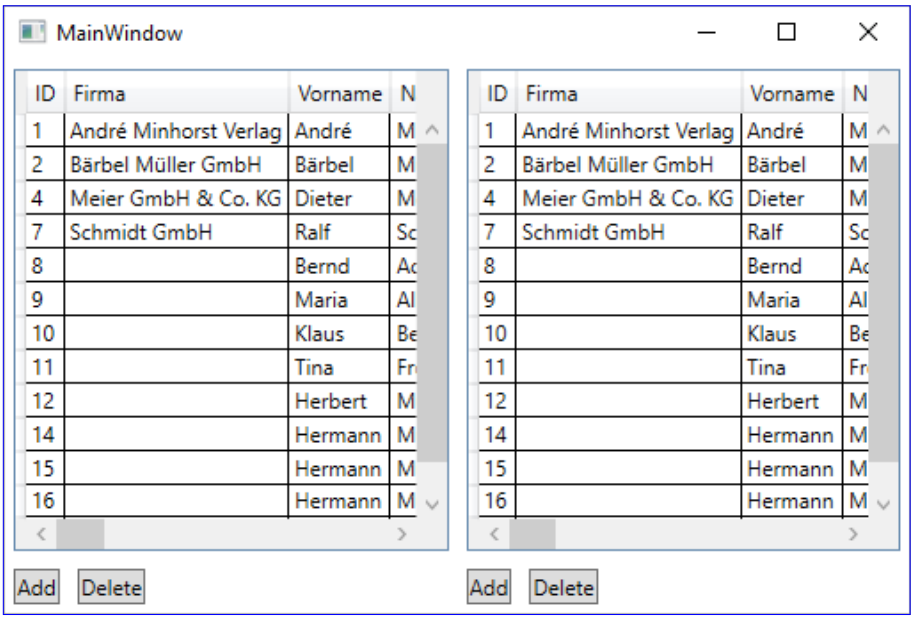


Bild 1: Beispiel für an ein **List**- und ein **ObservableCollection**-Objekt gebundene **DataGrid**-Steuerelemente

Was aber heißt überhaupt »informieren« in diesem Zusammenhang? Wenn Sie etwa unter Access ein Listenfeld an eine Tabelle gebunden und dann einen Datensatz dieser Tabelle gelöscht haben, wurde der Datensatz nach dem Aufruf der **Requery**-Methode des Listenfeldes auch aus dem Listenfeld entfernt. In einem Formular in der Datenblattansicht führte das Löschen eines Datensatzes direkt zum Entfernen dieses Eintrags aus der Ansicht – sehr praktisch, aber auch logisch, da hier direkt die Tabelle an das Formular gebunden war. Unter WPF ist das alles etwas anders – hier binden Sie ja beispielsweise nicht direkt an Tabellen

oder Abfragen, sondern an Auflistungen wie das **List**-Objekt, **Collection** oder **ObservableCollection**, die Sie zuvor noch mit den Daten aus der Datenbank füllen. Wenn Sie nun Änderungen an einem **List**- oder **Collection**-Objekt durchführen, wie es etwa geschieht, wenn Sie etwa einen neuen Eintrag hinzufügen oder einen Eintrag löschen, dann wirken sich die Änderungen zwar auf die Einträge des **List**- oder **Collection**-Objekts aus, aber sie schlagen sich nicht in der Benutzeroberfläche nieder.

Und hier tritt das **ObservableCollection**-Objekt auf den Plan: Wenn Sie diesem einen neuen Eintrag hinzufügen oder einen Eintrag entfernen, löst dies ein Ereignis aus, das automatisch von bestimmten Elementen der Benutzeroberfläche wie etwa dem **ListBox**- oder dem **DataGrid**-Element implementiert wird und dafür sorgt, dass die Ansicht aktualisiert wird. Um uns dies einmal an einem Beispiel anzusehen, haben wir das Fenster aus Bild 1 erstellt.

Die Definition dieser Steuerelemente finden Sie in Listing 1. Das erste DataGrid namens **dgList** ist an das Objekt **KundenList** der Code behind-Klasse gebunden, das zweite DataGrid namens **dgObservableCollection** an das Objekt **KundenObservable-Collection**. Zu jedem **DataGrid**-Steuerelement haben wir jeweils eine Hinzufügen- und eine Entfernen-Schaltfläche hinzugefügt, mit denen Sie per Code jeweils einen neuen Datensatz anlegen beziehungsweise den aktuell markierten Datensatz aus dem zugrunde liegenden Auflistungsobjekt entfernen. Wir werden dann später erkennen, wo die Unterschiede zwischen dem **List**- und dem **ObservableCollection**-Objekt liegen.

```
<Window x:Class="ObservableCollection.MainWindow" ...  Title="MainWindow" Height="350" Width="525">
    <Grid>
        <Grid.ColumnDefinitions>
            <ColumnDefinition></ColumnDefinition>
            <ColumnDefinition></ColumnDefinition>
        </Grid.ColumnDefinitions>
        <Grid.RowDefinitions>
            <RowDefinition Height="*"></RowDefinition>
            <RowDefinition Height="Auto"></RowDefinition>
        </Grid.RowDefinitions>
        <DataGrid x:Name="dgList" Grid.Column="0" Margin="5" ItemsSource="{Binding KundenList}"></DataGrid>
        <DataGrid x:Name="dgObservableCollection" Margin="5" Grid.Column="1"
            ItemsSource="{Binding KundenObservableCollection}"></DataGrid>
        <StackPanel Grid.Row="1" Grid.Column="0" Orientation="Horizontal">
            <Button x:Name="btnListAdd" Margin="5" Click="btnListAdd_Click">Add</Button>
            <Button x:Name="btnListDelete" Margin="5" Click="btnListDelete_Click">Delete</Button>
        </StackPanel>
        <StackPanel Grid.Row="1" Grid.Column="1" Orientation="Horizontal">
            <Button x:Name="btnObservableCollectionAdd" Margin="5" Click="btnObservableCollectionAdd_Click">Add</Button>
            <Button x:Name="btnObservableCollectionDelete" Margin="5"
                Click="btnObservableCollectionDelete_Click">Delete</Button>
        </StackPanel>
    </Grid>
</Window>
```

Listing 1: Definition der beiden **DataGrid**-Elemente und der Schaltflächen zum Hinzufügen und Löschen der Elemente

DataGrid-Elemente füllen

Damit die beiden **DataGrid**-Elemente gefüllt werden, haben wir der Code behind-Klasse etwas Code hinzugefügt. Im allgemeinen Teil haben wir dazu zunächst eine Objektvariable für den Datenbankkontext deklariert:

```
BestellverwaltungEntities dbContext;
```

Dann benötigen wir ein **List**-Objekt namens **kundenList**, das Elemente des Typs **Kunde** aufnehmen soll und über die öffentliche Eigenschaft **KundenList** per **get** und **set** verfügbar gemacht wird:

```
private List<Kunde> kundenList;
public List<Kunde> KundenList {
    get {
        return kundenList;
    }
    set {
        kundenList = value;
    }
}
```

Das Gleiche haben wir für ein Auflistungs-Element des Typs **ObservableCollection** durchgeführt. Dieses heißt allerdings **kundenObservableCollection** und wird wie folgt definiert:

```
private ObservableCollection<Kunde> kundenObservableCollection;
public ObservableCollection<Kunde> KundenObservableCollection {
    get {
        return kundenObservableCollection;
    }
    set {
        kundenObservableCollection = value;
    }
}
```

Damit Sie das **ObservableCollection**-Objekt überhaupt nutzen können, müssen Sie dieses zunächst über den passenden Namespace in der Klasse bekannt machen. Dies erledigen Sie mit der folgenden Zeile, die Sie zu den bestehenden **using**-Anweisungen hinzufügen:

```
using System.Collections.ObjectModel;
```

Damit kommen wir zur Konstruktor-Methode, die beim Anzeigen des Fensters ausgelöst wird. Sie erstellt das Datenbankkontext-Objekt **dbContext** und liest zunächst die Elemente der Auflistung **Kunden** des Datenbankkontexts in die Liste **kundenList** ein. Dann erledigt sie dies auch für die Liste **kundenObservableCollection**. Schließlich weist sie der Eigenschaft **DataContext**

einen Verweis auf die Code behind-Klasse selbst zu, damit die Elemente der Benutzeroberfläche die hier definierten Eigenschaften **KundenList** und **ObservableCollectionList** nutzen können, um die enthaltenen Daten anzuzeigen:

```
public MainWindow() {
    InitializeComponent();
    dbContext = new BestellverwaltungEntities();
    kundenList = dbContext.Kunden.ToList();
    kundenObservableCollection = new ObservableCollection<Kunde>(dbContext.Kunden);
    DataContext = this;
}
```

Element hinzufügen

Nun legen wir die Prozedur an, die durch das **Click**-Ereignis der ersten Schaltfläche **btnListAdd** ausgelöst wird. Diese erstellt ein neues **Kunde**-Objekt, weist diesem ein paar Eigenschaftswerte zu und fügt es dann als Parameter der **Add**-Methode zur Auflistung **kundenList** hinzu:

```
private void btnListAdd_Click(object sender, RoutedEventArgs e) {
    Kunde kunde = new Kunde();
    kunde.AnredeID = 1;
    kunde.Vorname = "Hermann";
    kunde.Nachname = "Müller";
    kundenList.Add(kunde);
}
```

Die Methode, die durch die Schaltfläche **btnObservableCollectionAdd_Click** ausgelöst wird, hat überwiegend identische Anweisungen, daher hier nur die gekürzte Fassung mit dem Aufruf der **Add**-Methode des **ObservableCollection**-Objekts **kundenObservableCollection**:

```
private void btnObservableCollectionAdd_Click(object sender,
RoutedEventArgs e) {
    ...
    kundenObservableCollection.
Add(kunde);
}
```

Beide Methoden erledigen grundsätzlich die gleichen Schritte für ihr

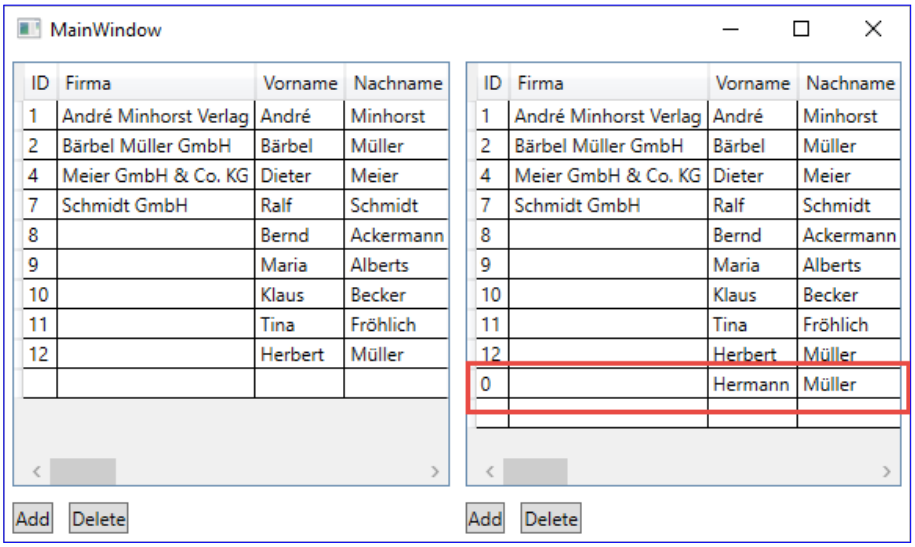


Bild 2: Nur in dem an das **ObservableCollection**-Objekt gebundene **DataGrid** taucht der neue Eintrag auf.

Type Converter in WPF/C#

Manche Eigenschaften eines .xaml-Dokuments sind sehr flexibel. Für die Eigenschaft Margin beispielsweise können Sie ganz verschiedenen Werte angeben – zum Beispiel einfach den Wert 5, um Abstände in alle vier Richtungen zu erhalten oder auch die Zeichenkette 5,0,5,0, um nur einen linken und einen rechten Abstand abzubilden. Auch andere Eigenschaften nehmen durchaus unterschiedliche Werte entgegen, zum Beispiel solche zur Angabe von Farben. Wenn man genauer hinsieht, stellt man schnell fest, dass man da Attributen, die völlig anderen Typs sind, ein String-Literal zuweist. Warum das hier gelingt, beschreibt der vorliegende Artikel.

Type Converter am Beispiel von Abständen

Ein Beispiel für die Angabe eines Strings für ein Attribut mit einem anderen Datentyp als String ist das Attribut **Margin**.

Wenn wir einmal einen Button nur mit einem Rand ausstatten, der laut unserer Angabe den Abstand **5** von der linken und rechten und den Abstand **10** von der oberen und unteren Kante des übergeordneten Elements aufweist, dann erhalten wir einen Button wie in Bild 1. Der Code sieht indes so aus:

```
<Button Margin="5,10,5,10">
```

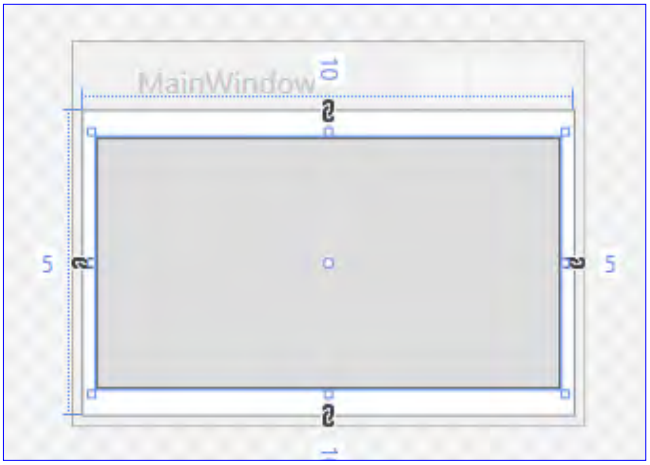


Bild 1: Ein **Button**-Element mit verschiedenen **Margin**-Werten

Wenn Sie im **.xaml**-Code einmal mit der rechten Maustaste auf das Attribut **Margin** klicken und dann den Eintrag **Gehe zu Definition** aus dem Kontextmenü auswählen, erscheint der Dialog aus Bild 2. Hier wird schnell deutlich, dass Margin keinesfalls den Datentyp **String** aufweist, sondern vielmehr den Datentyp **System.Windows.Thickness**. Wir müssten dieser Eigenschaft also normalerweise eine Struktur des Typs **Thickness** übergeben! Diese

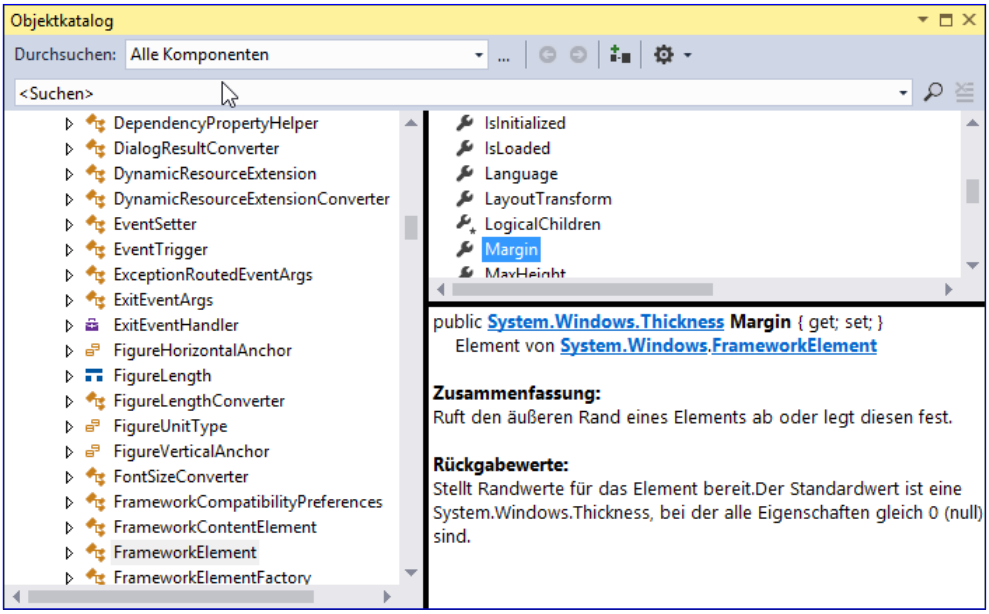


Bild 2: Definition der Eigenschaft **Margin**

Struktur weist wiederum die vier Eigenschaften **Bottom**, **Left**, **Right** und **Top** auf.

Margin unter C#

Um zu verdeutlichen, dass wir hier grundsätzlich einen nicht passenden Wert übergeben, wenn wir einen Zahlenwert oder auch einen String wie **5,10,5,10** angeben, schauen wir uns das Anlegen eines Buttons mit dieser Eigenschaft unter C# an.

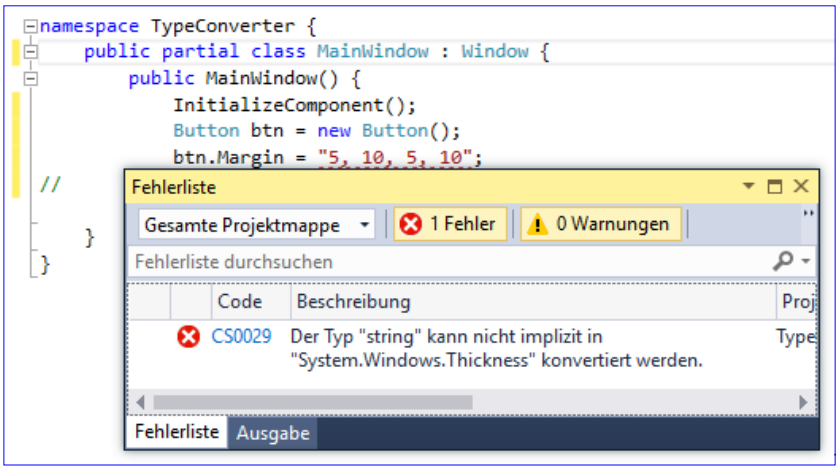


Bild 3: Unter C# kann der Eigenschaft **Margin** kein String übergeben werden.

Dazu erstellen wir mit der **new**-Anweisung ein neues Objekt des Typs **Button** und referenzieren dieses mit der Variablen **btn**. Danach weisen wir diesem genau wie in im **.xaml**-Code den Wert **„5, 10, 5, 10“** für die Eigenschaft **Margin** zu. Der gemeldete Fehler zeigt, dass **String** nicht in **Thickness** konvertiert werden kann (siehe Bild 3).

Was aber ist nun unter C# anders als im **.xaml**-Code? Der Unterschied ist, dass im **.xaml**-Code automatisch ein sogenannter Type Converter agiert und den angegebenen String in die gewünschten Werte umwandelt und zuweist.

Logisch, dass Sie hier nun auch nicht irgendeine Zeichenkette angeben können, sondern dass diese durchaus einen bis vier durch Kommata getrennte Zahlenwerte enthalten muss.

Wie geben wir die Werte nun unter C# ein? Dies erledigen wir, indem wir zunächst ein Objekt des Typs **Thickness** erstellen und seinen Eigenschaften **Bottom**, **Top**, **Left** und **Right** die gewünschten Zahlenwerte zuweisen.

Danach weisen wir der Eigenschaft **Margin** das Objekt **margin** mit dem Typ **Thickness** (damit wir den Button als Kindelement des Grids festlegen können, müssen wir diesem mit **x:Name=“MyGrid“** noch einen Namen zuweisen):

```
public partial class MainWindow : Window {
    public MainWindow() {
        InitializeComponent();
        Button btn = new Button();
        Thickness margin = new Thickness();
        margin.Bottom = 10;
        margin.Top = 10;
        margin.Left = 5;
        margin.Right = 5;
        btn.Margin = margin;
        MyGrid.Children.Add(btn);
    }
}
```

Fehlerhafte Bindungen prüfen

Wenn Sie mit C# programmieren, können Sie beim Debuggen bequem herausfinden, warum etwas nicht so funktioniert, wie Sie es sich vorstellen. Bei der Datenbindung von WPF-Elementen wird es komplizierter: Wenn etwa ein an ein Feld eines Objekts gebundenes Steuerelement nicht den gewünschten Wert anzeigt, kann dies verschiedene Gründe haben. Wenn das gebundene Element nicht gefunden werden kann, gibt es keine Fehlermeldung, und es wird auch schwierig, herauszufinden, woran es liegt. Dieser Artikel stellt ein paar Möglichkeiten vor, Licht ins Dunkel der Bindung zu bringen.

Verweis ins Leere

Manchmal ist man einfach betriebsblind oder unkonzentriert und gibt bei einer Bindung den Namen der falschen Eigenschaft ein. Dann bleibt das entsprechende Steuerelement schlicht leer! Wie aber finden Sie heraus, was genau nicht funktioniert? Der erste Ansatz ist: Die Ausgabe auf eventuelle Meldungen untersuchen. Nehmen wir den folgenden falschen Code zur Definition eines Grid-Elements mit Textfeldern – wie Sie sehen, kann schon ein Fehler bei der Groß-/Kleinschreibung zum Fehler führen:

```
<TextBox Grid.Column="1" Grid.Row="1" Text="{Binding Path=VorName}"></TextBox>
```

Als Ergebnis bleibt das Steuerelement leer, denn die Eigenschaft der an das Fenster gebundenen Klasse heißt nicht **VorName**, sondern **Vorname** (siehe Bild 1):

```
public class Kunde {
    public string Vorname { get; set; }
    ...
}
```

Bild 1: Das Feld **Vorname** bleibt erwartungsgemäß leer.

Nun stellen wir uns dumm und wollen herausfinden, warum das Feld keine Daten anzeigt. Eine Ausnahme ist auch nicht aufgetreten. Wertvolle Informationen liefert in diesem Fall das Ausgabe-Fenster (siehe Bild 2). Es liefert die folgende Meldung:

```
System.Windows.Data Error: 40 : BindingExpression path error: 'VorName' property not found on 'object' ''Kunde' (HashCode=16049999)'. BindingExpression:Path=VorName; DataItem='Kunde' (HashCode=16049999); target element is 'TextBox' (Name=''); target property is 'Text' (type 'String')
```

Hier erfahren wir also ziemlich genau, dass es die Eigenschaft **VorName** nicht im Objekt **Kunde** gibt.

Mehr Informationen ausgeben

Durch eine kleine Ergänzung des XAML-Codes können Sie noch eine Reihe weiterer Informationen im Ausgabefenster ausgeben. Dazu erweitern Sie zunächst das Window-Element um einen weiteren Namespace:

```
<Window x:Class="BindungenUntersuchen.MainWindow" xmlns="http://schemas.microsoft.com/winfx/2006/xaml/presentation"
```

Neuer Eintrag in ComboBox

Manchmal zeigen Kombinationsfelder einfach nur die verknüpften Werte einer Lookup-Tabelle an – wie beispielsweise bei der Kategorie eines Produktes. Die Tabelle mit den Produkten ist dann über ein Fremdschlüsselfeld mit der Lookup-Tabelle mit den Kategorien verknüpft, worüber dann die gewünschte Kategorie ausgewählt wird. Was aber, wenn man mal eben einen neuen Wert in die Lookup-Tabelle schreiben will? Lässt sich das überhaupt erledigen, ohne dass der Benutzer einen Dialog zur Eingabe der Daten der neuen Kategorie öffnet? Dieser Artikel bringt die Antwort auf diese Frage.

Unter Access ist dies ein sehr häufig genutztes Feature: Wenn ein Kombinationsfeld die Daten eines Lookup-Feldes zur Auswahl angeboten hat und der Benutzer einen neuen Eintrag zu der zugrunde liegenden Tabelle hinzufügen wollte, hat er einfach den gewünschten Wert in das Kombinationsfeld eingetragen und schon war der neue Datensatz mit den angegebenen Wert in der Tabelle gespeichert. Dazu gab es ein praktisches Ereignis namens **Bei nicht in Liste**, welches dann etwa mit Code gefüllt werden konnte, der den Benutzer fragte, ob der neue Eintrag tatsächlich in der Datenbank gespeichert und als neuer Wert genutzt werden sollte. Auch unter WPF und C# sollte es doch eine Möglichkeit geben, dies zu realisieren.

In unserer Beispielanwendung **Bestellverwaltung** haben wir dies bisher etwas komplizierter für den Benutzer gemacht. Die Daten der Tabelle **Kategorien** konnte der Benutzer für ein Produkt zwar komfortabel per Kombinationsfeld auswählen, aber wenn er einen neuen Eintrag zur Tabelle **Kategorien** hinzufügen wollte, müsste er schon noch einen eigenen Dialog über die entsprechende Ribbon-Schaltfläche aufrufen (siehe Bild 1).

Wir wollen nun sehen, ob wir das Kombinationsfeld so anpassen können, dass wir auch durch direkte Eingabe der Bezeichnung der Kategorie in das Kombinationsfeld einen neuen Eintrag zur Tabelle **Kategorien** hinzufügen können.

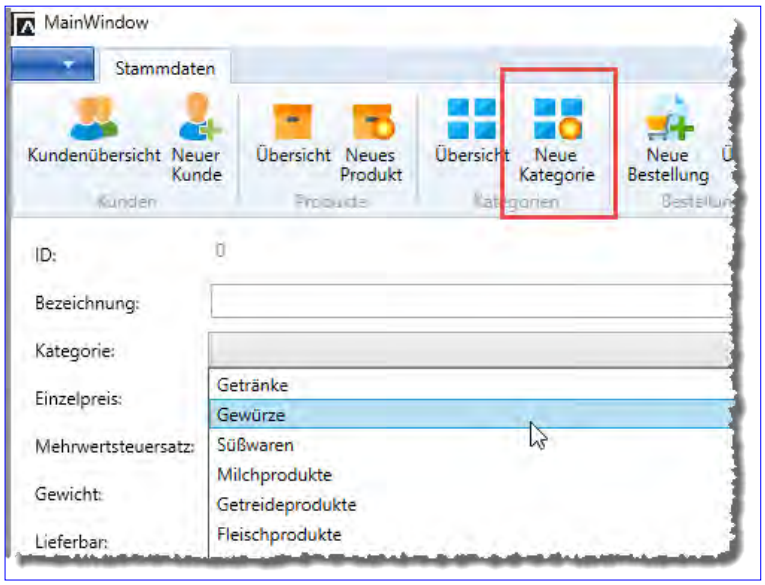


Bild 1: Neue Einträge fügt man über einen separaten Dialog hinzu.

Dazu fügen wir dem Ribbon eine Schaltfläche hinzu, mit der wir ein neues Fenster zum Anlegen eines neuen Produkts öffnen wollen. Wir könnten das auch mit dem bereits bestehenden **Page**-Objekt **Produktdetails.xaml** erledigen, aber ein eigenes Fenster ohne die Ereignisse und weiteren Code wird das Beispiel schlicht übersichtlicher. Das Fenster wollen wir zum Anzeigen der Details eines bestehenden Produkts und zum Anlegen eines neuen Produkts öffnen. Die Schaltflächen, welche das Fenster öffnen, bringen wir unten in der Seite **Produktuebersicht.xaml** unter. Die erste Schaltfläche heißt **btnNeuesProdukt_Window**, die zweite **btnProduktbearbeiten_Window**.

Also legen wir ein neues Fenster namens **Produktdetails_Kombi.xaml** an. Zur Definition dieses Fensters verwenden wir zunächst den gleichen **.xaml**-Code wie für das **Page**-Element **Produktdetails.xaml**:

```
<Window x:Class="Bestellverwaltung.Produktdetails_Kombi" ... Title="Produktdetails_Kombi" Height="300" Width="300">
    <Window.Resources>...</Window.Resources>
    <Grid x:Name="grid1" HorizontalAlignment="Stretch" Margin="8,8,0,0" VerticalAlignment="Top">
        //... Grid-Definition der Zeilen und Spalten ...
        <Label Content="ID:" Grid.Column="0" />
        <TextBox x:Name="txtID" Text="{Binding produkt.ID, Mode=TwoWay, ValidatesOnExceptions=true}" ... />
        <Label Content="Bezeichnung:" ... />
        <TextBox x:Name="txtBezeichnung" Text="{Binding produkt.Bezeichnung, Mode=TwoWay, ValidatesOnDataErrors=true}" ... />
        <Label Content="Kategorie:" ... />
        <ComboBox x:Name="cboKategorien" Grid.Column="1" HorizontalAlignment="Stretch" TabIndex="0"
                  Height="24" Grid.Row="2"
                  ItemsSource="{Binding Kategorien}"
                  SelectedItem="{Binding produkt.Kategorien, ValidatesOnDataErrors=True}"
                  DisplayMemberPath="Bezeichnung"
                  SelectedValuePath="ID"
                  VerticalAlignment="Center" Margin="1,4,29,4"></ComboBox>
        <Label Content="Einzelpreis:" ... />
        <TextBox x:Name="txtEinzelpreis" Text="{Binding produkt.Einzelpreis, Mode=TwoWay, ValidatesOnDataErrors=True}" ... />
        <Label Content="Mehrwertsteuersatz:" ... />
        <TextBox x:Name="txtMehrwertsteuersatz"
            Text="{Binding produkt.Mehrwertsteuersatz, Mode=TwoWay, ValidatesOnDataErrors=True}" ... />
        <Label Content="Gewicht:" ... />
        <TextBox x:Name="txtGewicht" Text="{Binding produkt.Gewicht, Mode=TwoWay, ValidatesOnDataErrors=True}" ... />
        <Label Content="Lieferbar:" ... />
        <CheckBox x:Name="chkLieferbar" IsChecked="{Binding produkt.Lieferbar, Mode=TwoWay, ValidatesOnDataErrors=True}" ... />
        <StackPanel Orientation="Horizontal" ... >
            <Button x:Name="btnSpeichern" Click="btnSpeichern_Click" Content="Speichern" ... />
            <Button x:Name="btnVerwerfen" Click="btnVerwerfen_Click" Content="Verwerfen" ... />
        </StackPanel>
    </Grid>
</Window>
```

Das Fenster sieht somit wie in Bild 2 aus. Um das Fenster zu öffnen, implementieren wir die beiden **Click**-Ereignisse für die auf der Seite **Produktuebersicht.xaml** neu angelegten Schaltflächen. Die Methode zum Öffnen des Formulars zum Anlegen eines neuen Produkts sieht wie folgt aus:

```
private void btnNeuesProdukt_Window_Click(object sender, RoutedEventArgs e) {
    Produktdetails_Kombi wnd = new Produktdetails_Kombi();
```

```
    wnd.ShowDialog();
}
```

Die Prozedur erstellt also einfach nur eine neue Instanz des Fensters **Produktdetails_Window** und öffnet dieses als modalen Dialog. Die Methode, die nach der Auswahl eines der Einträge der Produktübersicht und dem Anklicken der Schaltfläche **btnProduktAnzeigen** ausgelöst wird, ermittelt zunächst die **ID** des aktuell markierten Produkt-Datensatzes. Dann erstellt sie ebenfalls eine neue Instanz des Fensters **Produktdetail_Window** und übergibt die **ID** des markierten Produkts als Parameter für die Konstruktor-Methode:

Bild 2: Entwurf des Fensters **Produktdetails_Kombi.xaml**

```
private void btnProduktAnzeigen_Window_Click(object sender,
        RoutedEventArgs e) {
    Produkt produkt = (Produkt)dgProdukte.SelectedItem;
    int produktID = (int)produkt.ID;
    Produktdetails_Kombi wnd = new Produktdetails_Kombi(produktID);
    wnd.ShowDialog();
}
```

Damit steigen wir in den überschaubaren Code der Code behind-Klasse des Fensters **Produktdetails_Kombi** ein. Diese deklariert zunächst den Datenbankkontext für den Zugriff auf die Daten im Entity Data Model:

```
BestellverwaltungEntities dbContext = new BestellverwaltungEntities();
```

Außerdem benötigen wir ein Objekt für das anzuzeigende Produkt:

```
public Produkt produkt { get; set; }
```

Die Deklaration der Variablen zum Aufnehmen der Kategorien, die dann im Kombinationsfeld **cboKategorien** landen sollen, sieht hingegen etwas anders als im **Page**-Element **Produktdetails.xaml.cs** aus. Während wir dort noch ein einfaches **List**-Objekt verwendet haben, nutzen wir hier nun eine **ObservableCollection** sowohl für die private Variable als auch für die öffentliche Eigenschaft:

```
private ObservableCollection<Kategorie> kategorien;
public ObservableCollection<Kategorie> Kategorien {
    get { return kategorien; }
    set { kategorien = value; }
}
```

m:n-Beziehung mit Listenfeld

Wir haben in der Beispieldatenbank Bestellverwaltung bereits eine m:n-Beziehung über ein Fenster realisiert, das an eine Tabelle gebunden ist und ein DataGrid enthält, das die verknüpfte Elemente anzeigt. Dabei handelt es sich um die Abbildung von Bestellungen, Bestelldetails und Produkte, wo zusätzlich zur Verknüpfung noch weitere Daten wie der Einzelpreis für die Bestellposition gespeichert werden. Im vorliegenden Artikel werden wir uns ansehen, wie wir eine m:n-Beziehung ohne weitere Daten in der Verknüpfungstabelle verwalten können. Dazu wollen wir Kunden über eine Verknüpfungstabelle einer Tabelle mit Versendungen etwa zu Werbe- oder Informationszwecken verknüpfen.

Notwendige Tabellen/Erweiterungen

Die Datenbank **Bestellverwaltung.db** auf Basis von SQLite, die wir in den bisherigen Beispielen verwendet haben, müssen wir für diesen Artikel um zwei Tabellen erweitern.

Bild 1: Anlegen der neuen Tabelle **Versendungen**

Das erledigen wir schnell über das Verwaltungsprogramm SQLite Studio, mit dem wir die Datei **Bestellverwaltung.db** öffnen und als Erstes die Tabelle **Versendungen** anlegen – und zwar mit den Feldern wie in Bild 1. Speichern Sie die Tabelle nach dem Anlegen der Felder und des Tabellennamens mit einem Klick auf die Schaltfläche **Commit structure changes**.

Danach folgt die zweite Tabelle **KundenVersendungen** als Verknüpfungstabelle. Diese Tabelle enthält neben dem Primärschlüsselfeld **ID** noch die beiden Fremdschlüsselfelder **KundeID** und **VersendungID**, welche die beiden Tabellen **Kunden** und **Versendungen** referenzieren.

Wenn Sie die Fremdschlüsselfelder anlegen, aktivieren Sie im Dialog **Spalte** die Option **Fremdschlüssel** und klicken dann auf die Schaltfläche **Konfigurieren**. Dies zeigt den Dialog

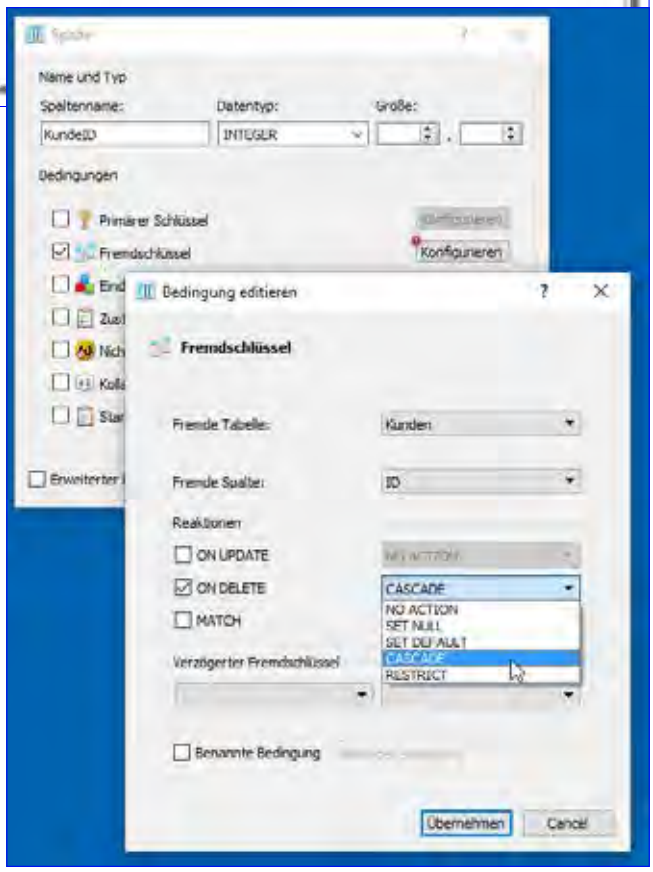


Bild 2: Anlegen der neuen Tabelle **KundenVersendungen**

Bedingung editieren an, wo Sie als **Fremde Tabelle** die Tabelle **Kunden** und als **Fremde Spalte** die Spalte **ID** auswählen (siehe Bild 2). Hier aktivieren Sie außerdem die Option **ON DELETE** und wählen dafür die Aktion **CASCADE** aus.

Dies bewirkt, dass wenn Sie einen Datensatz der Tabelle **Kunden löschen**, für den in der Tabelle **KundenVersendungen** verknüpfte Datensätze vorliegen, diese auch gelöscht werden. Das Gleiche führen wir auch für das Fremdschlüsselfeld zum Verknüpfen mit der Tabelle **Versendungen** durch. Speichern Sie die Tabelle mit der Schaltfläche **Commit structure changes**.

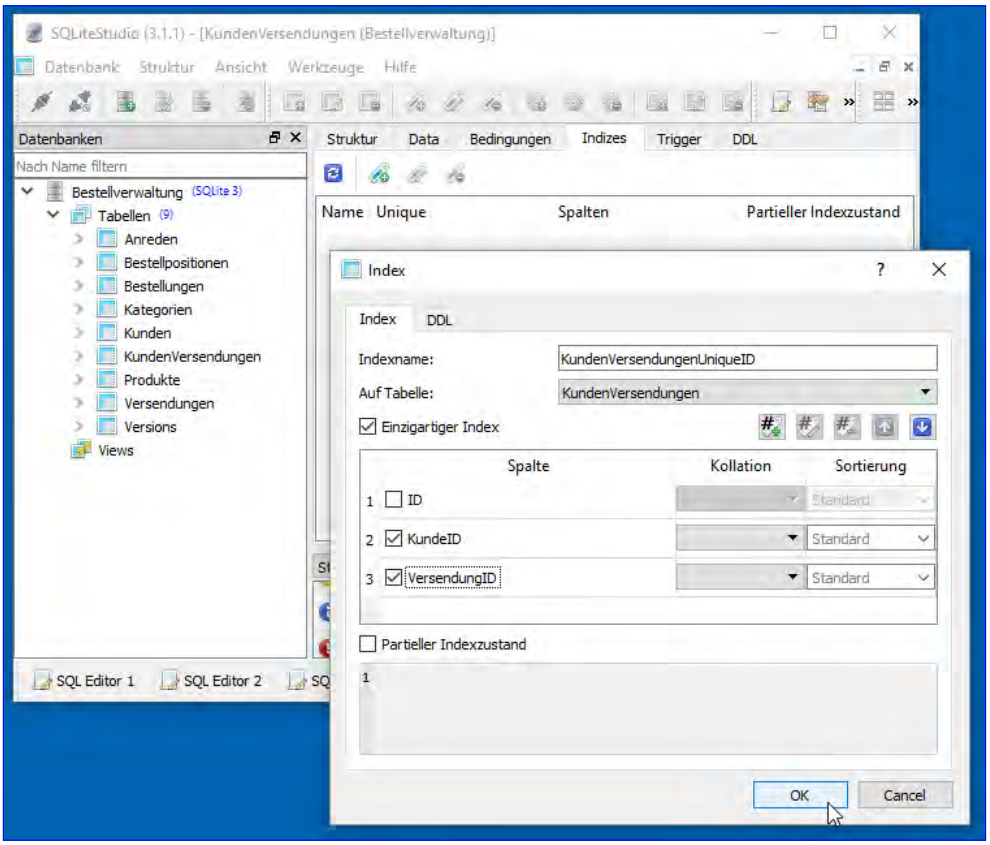


Bild 3: Anlegen eines zusammengesetzten, eindeutigen Schlüssels über die beiden Felder **KundeID** und **VersendungID**

Danach legen wir noch einen zusammengesetzten, eindeutigen Schlüssel für die beiden Fremdschlüsselfelder **KundeID** und **VersendungID** an, damit jede Versendung jedem Kunden nur einmal zugeordnet werden kann. Dazu klicken Sie oben über dem Tabellenentwurf auf den Reiter **Indizes**.

Klicken Sie dann auf die Schaltfläche **Create index (ins)**. Dann tragen Sie im nun erscheinenden Dialog **Index** den Wert **KundenVersendungenUniqueIndex** als Name ein. Wählen Sie die beiden Felder **KundeID** und **VersendungID** aus und aktivieren Sie die Option **Einzigartiger Index** (okay, die Übersetzung ist zumindest gut gemeint von den Entwicklern des Tools). Dies soll dann wie in Bild 3 aussehen, bevor Sie auf **OK** klicken und anschließend das Anlegen des Indizes bestätigen.

Änderungen im Entity Data Model

Nachdem Sie die beiden neuen Tabellen angelegt haben, benötigen wir in der Anwendung die entsprechenden Entitäten für die beiden Tabellen. Dazu öffnen Sie die Anwendung **Bestellverwaltung** und klicken im Projekt-Explorer doppelt auf das Objekt **Bestellverwaltung.edmx**.

Klicken Sie mit der rechten Maustaste in das Diagramm und wählen Sie aus dem Kontextmenü den Eintrag **Modell aus der Datenbank aktualisieren...** aus. Gegebenenfalls müssen Sie nun die Verbindung für das Entity Data Model festlegen. Danach

erscheint dann der Dialog, der auch beim Erstellen eines Entity Data Models angezeigt wird. Hier wählen Sie die beiden einzigen angezeigten Tabellen **KundenVersendungen** und **Versendungen** aus und klicken auf **Fertig stellen**.

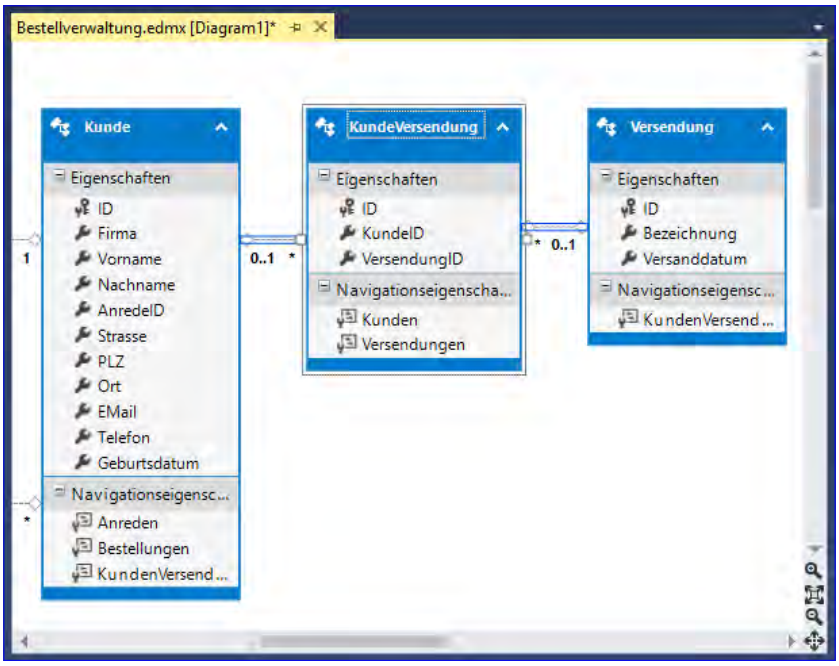


Bild 4: Die beiden neuen Entitäten im Entity Data Model, hier bereits in der Singular-Benennung

Danach sollten die neuen Tabellen wie in Bild 4 im Entity Data Model angezeigt werden, sodass wir nun darauf zugreifen können. Genau wie bei den übrigen Entitäten haben wir auch bei diesen beiden Entitäten die Benennung in den Singular geändert, also von **KundenVersendungen** auf **KundeVersendung** und von **Versendungen** auf **Versendung**.

Wenn Sie nun noch den neuen Zustand etwa mit der Tastenkombination **Strg + S** speichern, werden auch die dahinter liegenden Objekte umbenannt.

Fenster zum Zuordnen von Kunden und Versendungen

Das Fenster, in dem wir einer Versendung die entsprechenden Kunden zuordnen wollen, soll im oberen Bereich ein Kombinationsfeld enthalten, welches die Versendungen abbildet. Im unteren Bereich wollen wir zwei Listenfelder darstellen. Das linke enthält alle Kunden, die der Versendung bereits zugeordnet wurden und das rechte als übrigen Kunden. Dazwischen platzieren wir vier Schaltflächen für die folgenden Funktionen:

- **<**: Den aktuell markierten Kunden aus dem rechten Listenfeld zur Liste der Empfänger hinzufügen.
- **>**: Den aktuell markierten Kunden aus dem linken Listenfeld aus der Liste der Empfänger entfernen.
- **<<**: Alle Kunden aus dem rechten Listenfeld zur Liste der Empfänger hinzufügen.

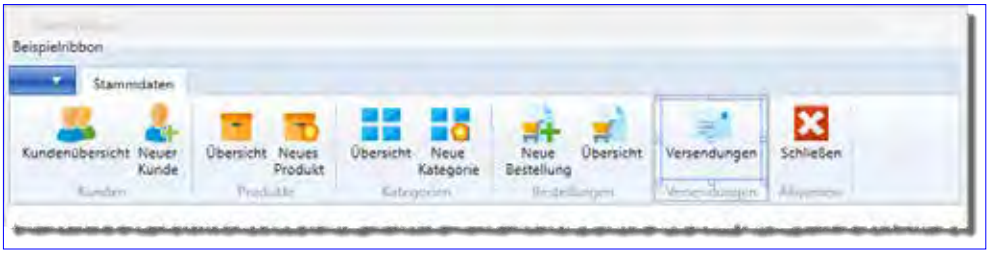


Bild 5: Neue Ribbon-Schaltfläche zum Anzeigen des Fensters mit den Zuweisungen der Versendungen

- **>>**: Alle Kunden aus dem linken Listenfeld aus der Liste der Empfänger entfernen.

Fenster anlegen

Das neue Fenster wollen wie diesmal einmal nicht in die bestehende Struktur einbinden, sondern über den Ribbon-Eintrag **Versendungen** als eigenes Fenster öffnen. Das Fenster legen Sie im Projekt unter dem Namen **Versendungen.xaml** an. Dem Fenster **MainWindow.xaml** fügen wir eine neue Ribbon-Schaltfläche wie in Bild 5 hinzu. Der Code dafür sieht wie folgt aus:

```
<RibbonGroup Header="Versendungen">
    <RibbonButton Name="btnVersendungen" Label="Versendungen" Click="btnVersendungen_Click"
        LargeImageSource="images\mail2.png"></RibbonButton>
</RibbonGroup>
```

Für die Schaltfläche hinterlegen wir die folgende Ereignismethode:

```
private void btnVersendungen_Click(object sender,
        RoutedEventArgs e) {
    Versendungen wnd = new Versendungen();
    wnd.ShowDialog();
}
```

Damit schauen wir uns nun den Entwurf des Fensters **Versendungen.xaml** an (siehe Bild 6). Das Fenster enthält drei Spalten und drei Zeilen. Die erste Zeile enthält lediglich das Kombinationsfeld **cboVersendungen** zur Auswahl der gewünschten Versendung. Der Inhalt befindet sich in einem horizontal ausgerichteten **StackPanel**-Element, dass sich über alle drei Spalten erstreckt (**Grid.ColumnSpan="3"**).

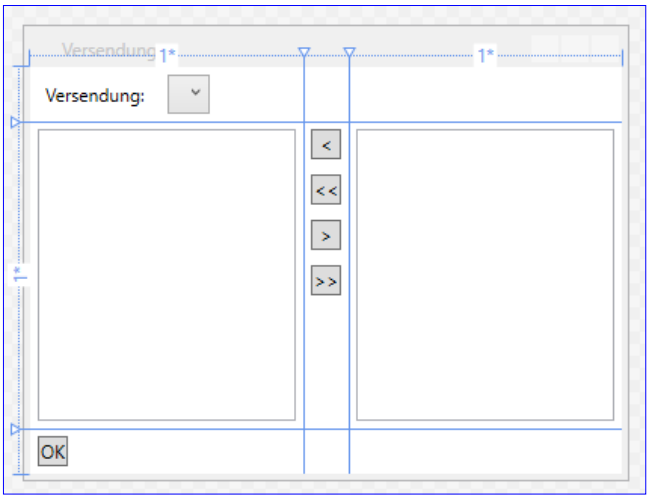


Bild 6: Entwurf des Fensters **Versendungen.xaml**

Die zweite Zeile enthält in der ersten Spalte das erste Listenfeld, in der zweiten Spalte ein vertikales **StackPanel**-Element mit den vier Schaltflächen und die dritte Spalte das zweite Listenfeld. Die dritte Spalte schließlich enthält eine **OK**-Schaltfläche zum Schließen des Fensters. Der **.xaml**-Code des Fensters sieht in gekürzter Form wie folgt aus:

```
<Window x:Class="Bestellverwaltung.Versendungen" ... Title="Versendungen" Height="300" Width="400">
    <Grid>
        <Grid.RowDefinitions>
            <RowDefinition Height="Auto"></RowDefinition>
            <RowDefinition Height="*"></RowDefinition>
            <RowDefinition Height="Auto"></RowDefinition>
```

```
        </Grid.RowDefinitions>
        <Grid.ColumnDefinitions>
            <ColumnDefinition></ColumnDefinition>
            <ColumnDefinition Width="Auto"></ColumnDefinition>
            <ColumnDefinition></ColumnDefinition>
        </Grid.ColumnDefinitions>
        <StackPanel Orientation="Horizontal" Grid.Row="0" Grid.Column="0" Grid.ColumnSpan="3">
            <Label Margin="5">Versendung:</Label>
            <ComboBox x:Name="cboVersendungen" Margin="5"></ComboBox>
        </StackPanel>
        <ListBox Grid.Column="0" Grid.Row="1" Margin="5"></ListBox>
        <StackPanel Orientation="Vertical" Grid.Column="1" Grid.Row="1">
            <Button x:Name="btnEinenHinzufuegen" Margin="5" Content="&lt;"></Button>
            <Button x:Name="btnAlleHinzufuegen" Margin="5" Content="&lt;&lt;"></Button>
            <Button x:Name="btnEinenEntfernen" Margin="5" Content="&gt;"></Button>
            <Button x:Name="btnAlleEntfernen" Margin="5" Content="&gt;&gt;"></Button>
        </StackPanel>
        <ListBox Grid.Column="2" Grid.Row="1" Margin="5"></ListBox>
        <StackPanel Orientation="Horizontal" Grid.Row="2" Grid.Column="0" Grid.ColumnSpan="3">
            <Button x:Name="btnOK" Margin="5" Click="btnOK_Click">OK</Button>
        </StackPanel>
    </Grid>
</Window>
```

Interessant ist hier, wie wir die Breite und die Höhe der Spalten und Zeilen eingestellt haben. Damit die mittlere Spalte genau die für die vier Schaltflächen in dem Stackpanel benötigte Spaltenbreite annimmt, haben wir die Eigenschaft **Width** auf den Wert **Auto** eingestellt. Die linke und die rechte Spalte sollen sich den übrigen Platz gleichmäßig aufteilen, daher erhält die Eigenschaft **Width** hier den Wert *.

Die obere Zeile und die untere Zeile sollen wie die mittlere Spalte nur den benötigten Platz einnehmen, also stellen wir die Eigenschaft **Height** für beide auf den Wert **Auto** ein. Die mittlere Zeile soll den Rest der Höhe beanspruchen, war wir durch den Wert * für die Eigenschaft **Height** erreichen.

Die Definition enthält bislang nur die reinen Steuerelemente ohne jegliche Bindung.

Kombinationsfeld mit den Versendungen füllen

Um das Kombinationsfeld mit den Datensätzen der Tabelle **Versendungen** in alphabetischer Reihenfolge zu füllen, müssen zunächst im Code behind-Modul eine entsprechende Auflistung erzeugen und aus dem Entity Data Model füllen.

Dazu erstellen wir zunächst einen Datenbank-Kontext im allgemeinen Teil der Klasse, der somit direkt beim Erstellen angelegt wird:

```
BestellverwaltungEntities dbContext = new BestellverwaltungEntities();
```

Für den Inhalt des Kombinationsfeldes legen wir eine ObservableCollection an, die wir als private Variable namens **versendungen** sowie als öffentliche Variable namens **AlleVersendungen** definieren (**AlleVersendungen**, weil wir bereits das Fenster **Versendungen** genannt haben):

```
private ObservableCollection<Versendung> versendungen;
public ObservableCollection<Versendung> AlleVersendungen{
    get { return versendungen; }
    set { versendungen = value; }
}
```

In der Konstruktor-Methode des Fensters füllen wir dann zunächst die ObservableCollection, die als Datenquelle für das Kombinationsfeld dienen soll und weisen dem Fenster die Code behind-Klasse als **DataContext** zu:

```
public Versendungen() {
    InitializeComponent();
    versendungen = new ObservableCollection<Versendung>(dbContext.Versendungen);
    DataContext = this;
}
```

Nun passen wir in **Versendungen.xaml** die Definition des **ComboBox**-Elements an, damit es die Daten der **ObservableCollection** anzeigt:

```
<ComboBox x:Name="cboVersendungen" ... IsEditable="True"
    Text="{Binding NewEntry, UpdateSourceTrigger=LostFocus}"
    ItemsSource="{Binding AlleVersendungen}"
    DisplayMemberPath="Bezeichnung"
    SelectedValuePath="ID"
    VerticalAlignment="Center"></ComboBox>
```

Hier haben wir gleich die Vorbereitungen getroffen, dass der Benutzer neue Versendungen direkt in das Kombinationsfeld eingeben kann. Wie das im Detail funktioniert, erläutern wir im Artikel **Neuer Eintrag in ComboBox**. Hier nur die Kurzfassung: **IsEditable="True"** sorgt dafür, dass der Benutzer Text in das Kombinationsfeld eintippen kann. Das Binding für das Attribut **Text** sorgt dafür, dass eine Eigenschaftsmethode namens **NewEntry** neue Werte entgegennimmt und **UpdateSourceTrigger="LostFocus"** sorgt dafür, dass die Methode **NewEntry** beim Fokusverlust ausgelöst wird. Die Methode **NewEntry** sieht so aus:

```
public string NewEntry {
    set {
        if (cboVersendungen.SelectedIndex == -1) {
```

```
            if (!string.IsNullOrEmpty(value)) {
                MessageBoxResult result = MessageBox.Show("Eintrag '" + value + "' zu den Versendungen hinzufügen?",
                    "Neuer Eintrag", MessageBoxButton.YesNo);
                if (result == MessageBoxResult.Yes) {
                    Versendung newVersendung = new Versendung();
                    newVersendung.Bezeichnung = value;
                    versendungen.Add(newVersendung);
                    dbContext.Versendungen.Add(newVersendung);
                    dbContext.SaveChanges();
                }
            }
        }
    }
}
```

Sie prüft, ob der Inhalt des eingegebenen Textes, der wie üblich über den Parameter **value** geliefert wird, nicht leer ist und fragt in diesem Fall den Benutzer, ob er den Wert als neuen Eintrag hinzufügen möchte (siehe Bild 7). Falls ja, legt die Methode ein neues Objekt des Typs **Versendung** an und weist **value** als **Bezeichnung** hinzu. Dann fügt sie das neue Objekt zur **ObservableCollection** und zur entsprechenden Liste des Entity Data Models hinzu und speichert die Änderung in der Datenbank.

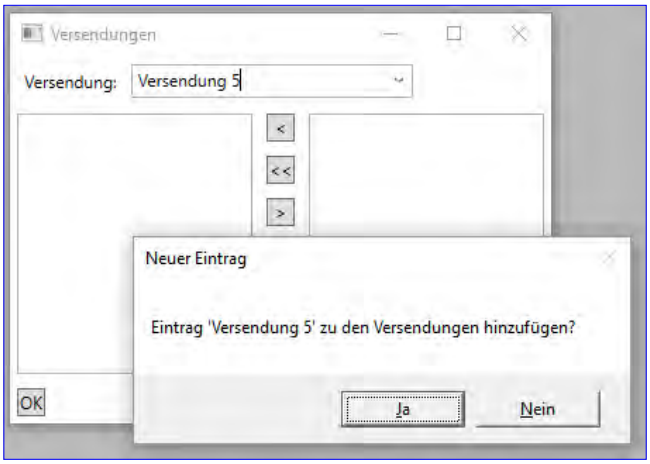


Bild 7: Anlegen eines neuen Eintrags im Kombinationsfeld

Listenfelder füllen

Nun wollen wir die Listenfelder mit den Kunden füllen, die einer Versendung zugewiesen wurden oder auch nicht. Dabei kommt die m:n-Beziehung zwischen den Versendungen und den Kunden ins Spiel. Das linke Listenfeld soll alle Kunden anzeigen, die über die m:n-Verknüpfungstabelle **KundenVersendungen** der aktuell im Kombinationsfeld **cboVersendungen** zugeordnet sind, das rechte Listenfeld alle Kunden, die der Versendung nicht zugeordnet sind. Diese Anzeige soll jeweils beim Auswählen eines neuen Eintrags im Kombinationsfeld aktualisiert werden. Außerdem sorgen wir gleich noch dafür, dass beim Öffnen des Fensters der erste Eintrag des Kombinationsfeldes vorausgewählt wird – und damit auch die entsprechenden Einträge in den Listenfeldern. Die Konstruktor-Methode erweitern wir also wie folgt:

```
public Versendungen() {
    InitializeComponent();
    versendungen = new ObservableCollection<Versendung>(dbContext.Versendungen);
    Versendung versendung = versendungen.First();
    cboVersendungen.SelectedIndex = 0;
```

LINQ to Entities-Beispiele in der Methodensyntax

LINQ to Entities ist die Abfragesprache für den Zugriff auf Daten in den Entitäten eines Entity Data Models. Im vorliegenden Artikel sehen wir uns einige Beispiele für den Zugriff per LINQ auf die Daten unserer Beispielanwendung »Bestellverwaltung« an. Dabei verwenden wir in diesem Artikel die Methoden-Syntax von LINQ.

Voraussetzungen

Für die hier vorgestellten Beispiele verwenden wir das Entity Data Model der Anwendung **Bestellverwaltung**. Die Ergebnisse sollen jeweils in einem DataGrid ausgegeben werden. Dazu erstellen wir ein neues Fenster namens **LINQBeispiele**, dessen Entwurf wie in Bild 1 aussieht. Die Definition ist einfach gehalten. Die wichtigsten Elemente ist das Kombinationsfeld **cboBeispiele**, mit dem Sie die Beispiele auswählen können, die wir anschließend per C#-Code zum Code behind-Modul hinzufügen, sowie das **DataGrid**-Steuerelement **db**, welches das Ergebnis der jeweiligen Abfrage anzeigen soll. Das **ComboBox**-Steuerelement ist an die Eigenschaft **Beispiele** gebunden und löst das Ereignis **SelectionChanged** aus:

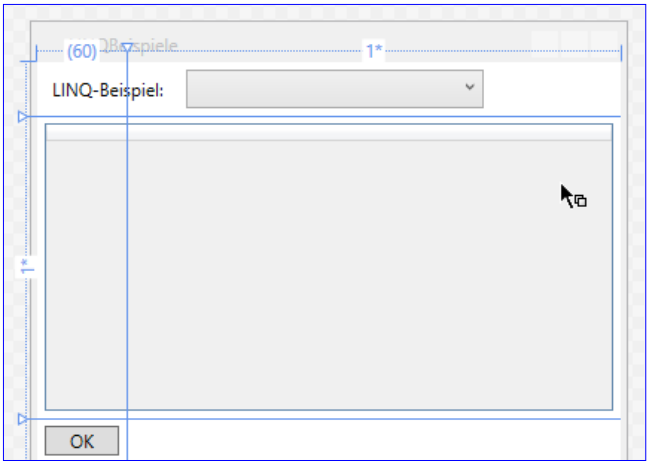


Bild 1: Fenster zur Ausgabe der Beipiele

```
<Window x:Class="Bestellverwaltung.LINQBeispiele"
        Title="LINQBeispiele" Height="300" Width="400">
    <Grid>
        ... Grid-Definition ...
        <StackPanel Grid.Column="0" Grid.ColumnSpan="2" Orientation="Horizontal">
            <Label Margin="5">LINQ-Beispiel:</Label>
            <ComboBox x:Name="cboBeispiele" Margin="5" Width="200" SelectionChanged="cboBeispiele_SelectionChanged"
                ItemsSource="{Binding Beispiele}"></ComboBox>
        </StackPanel>
        <DataGrid x:Name="dg" Grid.Row="1" Grid.Column="0" Grid.ColumnSpan="2" Margin="5"></DataGrid>
        <Button x:Name="btnOK" Grid.Row="2" Margin="5" Width="50" Click="btnOK_Click">OK</Button>
    </Grid>
</Window>
```

In der Code behind-Klasse erstellen wir ein Datenbank-Kontext-Objekt mit Zugriff auf das Entity Data Model sowie eine **ObservableCollection**, welche die Liste der Beispiele für das Kombinationsfeld enthält:

```
BestellverwaltungEntities dbContext = new BestellverwaltungEntities();
private ObservableCollection<string> beispiele;
```

```
public ObservableCollection<string> Beispiele {
    get { return beispiele; }
    set { beispiele = value; }
}
```

In der Konstruktor-Methode erstellen wir die **ObservableCollection** und füllen diesem mit Einträgen wie hier am Beispiel **Alle Kunden**:

```
public LINQBeispiele() {
    InitializeComponent();
    beispiele = new ObservableCollection<string>();
    beispiele.Add("Alle Kunden");
    DataContext = this;
}
```

Für das konkrete Beispiel, alle Kunden anzuzeigen, definieren wir auch wieder eine **ObservableCollection**:

```
private ObservableCollection<Kunde> kunden;
public ObservableCollection<Kunde> Kunden {
    get { return kunden; }
    set { kunden = value; }
}
```

Diese wird schließlich gefüllt und angezeigt, wenn der Benutzer den Eintrag **Alle Kunden** aus dem Kombinationsfeld auswählt. Dies löst das Ereignis **SelectionChanged** aus, für das wir die folgende Ereignismethode hinterlegt haben:

```
private void cboBeispiele_SelectionChanged(object sender, SelectionChangedEventArgs e) {
    ComboBox cbo = (ComboBox)sender;
    string beispiel = (string)cbo.SelectedItem;
    switch (beispiel) {
        case "Alle Kunden":
            kunden = new ObservableCollection<Kunde>(dbContext.Kunden);
            dg.ItemsSource = Kunden;
            break;
        default:
            break;
    }
}
```

Für den Eintrag **Alle Kunden** füllt die Methode nun die **ObservableCollection** namens **kunden** mit den über den LINQ-Ausdruck **dbContext.Kunden** ermittelten Kunden. Die Eigenschaft **Kunden** wird dann noch dem DataGrid **dg** zugewiesen, damit

dieses die Kundenliste wie in Bild 2 anzeigt. Das DataGrid ist ja flexibel, das heißt, Sie können ihm beliebige Auflistungen zuweisen. Wir fügen daher, wenn wir in den folgenden Beispielen andere **ObservableCollections** als **Kunden** nutzen, einfach entsprechende neue ObservableCollections hinzu und weisen die entsprechende Collection der Eigenschaft **ItemsSource** des **DataGrid**-Elements zu.

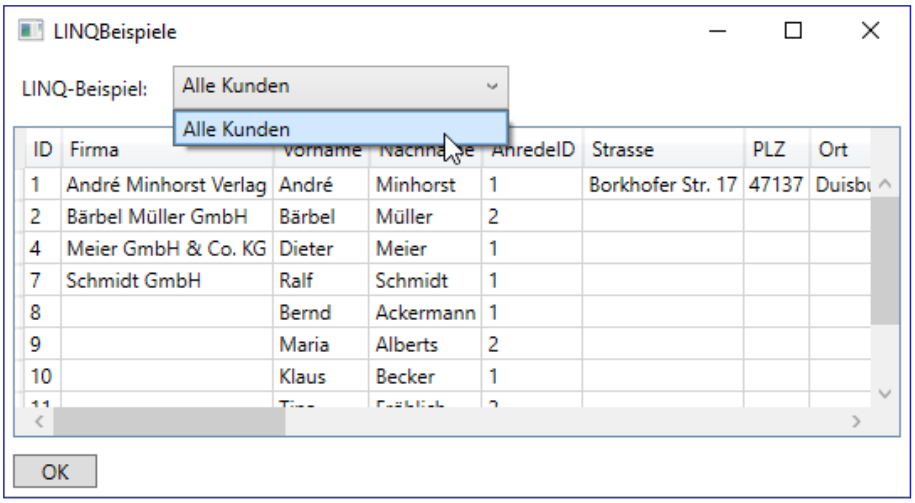


Bild 2: Ausgabe aller Kunden des Entity Data Models

Alle Elemente einer Klasse

Das erste Beispiel haben Sie ja schon kennengelernt. Um alle Elemente einer Klasse beziehungsweise Auflistung zu liefern, brauchen Sie diese einfach nur gemeinsam mit dem **dbContext**-Objekt als Inhalt der neuen ObservableCollection anzugeben:

```
kunden = new ObservableCollection<Kunde>(dbContext.Kunden);
```

Dies liefert alle Elemente der **Kunden**-Auflistung des Entity Data Models. Alle Elemente von anderen Klassen können Sie ebenso leicht holen, dazu müssen Sie nur die entsprechende **ObservableCollection** definieren und dem DataGrid zuweisen. Wenn Sie nun also alle Produkte ausgeben wollen, legen Sie erst eine neue ObservableCollection samt privater und öffentlicher Variable an:

```
private ObservableCollection<Produkt> produkte;
public ObservableCollection<Produkt> Produkte {
    get { return produkte; }
    set { produkte = value; }
}
```

Dann fügen Sie einen neuen Eintrag zur **ComboBox** hinzu:

```
public LINQBeispiele() {
    ...
    beispiele.Add("Alle Produkte");
    ...
}
```

Schließlich noch der neue **case**-Zweig in der **switch**-Bedingung:

```
case "Alle Produkte":
    produkte = new ObservableCollection<Produkt>(dbContext.Produkte);
```

```
    dg.ItemsSource = Produkte;
    break;
```

Damit können wir uns nun in den folgenden Beispielen auf die LINQ-Ausdrücke konzentrieren.

Sortieren von Elementen

Zum Sortieren nutzen wir die **OrderBy**- oder die **OrderByDescending**-Methode. Letztere ist für die absteigende Sortierung verantwortlich. Der folgende Ausdruck liefert alle Kunden, aufsteigend sortiert nach den Nachnamen:

```
dbContext.Kunden.OrderBy(d => d.Nachname)
```

Wenn Sie die Kunden absteigend nach den Nachnamen sortieren wollen, verwenden Sie **OrderByAscending**:

```
dbContext.Kunden.OrderByDescending(d => d.Nachname)
```

Für Sortierungen nach verschiedenen Feldern in der gleichen Abfrage verwenden Sie **OrderBy**/**OrderByDescending**- gefolgt von einem oder mehreren Aufrufen der **ThenBy**/**ThenByDescending**-Methode. Wenn Sie also zuerst nach dem Nachnamen und dann nach dem Vornamen sortieren wollen, verwenden Sie den folgenden Ausdruck:

```
dbContext.Kunden.OrderBy(d => d.Nachname).ThenBy(d => d.Vorname)
```

Man könnte fälschlicherweise meinen, zwei aufeinanderfolgende Aufrufe der **OrderBy**-Methode würden die Aufgabe auch erledigen. Das ist aber mitnichten der Fall. Die folgende Abfrage sortiert nämlich erst nach dem Vornamen und dann nach dem Nachnamen – allerdings wird die erste Sortierung beim Ausführen der zweiten verworfen, sodass auch bei mehreren gleichen Nachnamen die Datensätze nur zufällig richtig nach dem Vornamen sortiert sein dürften:

```
dbContext.Kunden.OrderBy(d => d.Vorname).OrderBy(d => d.Nachname)
```

Der folgende Ausdruck sortiert aufsteigend nach den Nachnamen und absteigend nach dem Vornamen:

```
dbContext.Kunden.OrderBy(d => d.Nachname).ThenByDescending(d => d.Vorname)
```

Filtern nach einzelnen Feldern

Wenn Sie Daten etwa nach dem Inhalt eines Feldes filtern wollen, können Sie das mit der **Where**-Methode erledigen. Das folgende Beispiel liefert etwa alle Kunden, deren Feld **Nachname** den Wert **Müller** enthält:

```
dbContext.Kunden.Where(d => d.Nachname == "Müller")
```

Wenn Sie alle Datensätze wünschen, deren Nachname mit dem Buchstaben **M** beginnt, ist dies der richtige Ausdruck:

```
dbContext.Kunden.Where(d => d.Nachname.StartsWith("M"))
```

Datensätze, deren Nachname den Buchstaben **m** enthält:

```
kunden = new ObservableCollection<Kunde>(dbContext.Kunden.Where(d => d.Nachname.Contains("m")));
```

Datensätze, deren Nachname auf **n** endet:

```
kunden = new ObservableCollection<Kunde>(dbContext.Kunden.Where(d => d.Nachname.EndsWith("n")));
```

Vergleichen mit Datumsangaben

Kunden, die älter als von 1976 sind:

```
dbContext.Kunden.Where(d => d.Geburtsdatum < new DateTime(1976,1,1))
```

Kunden, die im Jahr 1976 geboren wurden. Achtung: hier müssen Sie zwar für das erste Datum, also **1.1.1976** den Vergleichsoperator **>=** verwenden. Für die Obergrenze, also **1.1.1977**, verwenden sie jedoch den Operator **<**. Hier greifen wir etwas vor und nutzen schon den **Und**-Operator für die Kombination zweier Bedingungen:

```
dbContext.Kunden.Where(d => d.Geburtsdatum >= new DateTime(1976, 1, 1) && d.Geburtsdatum < new DateTime(1977,1,1))
```

Und- und Oder-Verknüpfung in der Bedingung

Wenn die Bedingung mehrere Elemente enthält, die mit Und oder Oder verknüpft werden sollen, verwenden Sie für **Und** den Operator **&&** und für **Oder** den Operator **||**.

Verknüpfte Tabellen, 1:n

Da wir in den bisherigen Beispielen immer die Daten einer Entität als Einheit betrachtet haben und nicht wie etwa unter Access per Abfrage die Daten mehrerer Tabellen zusammenfassen und anzeigen, wollen wir auch bei der Formulierung der LINQ to Entities-Abfragen beim Zugriff auf Entitäten bleiben. Allerdings wollen wir durchaus verknüpfte Entitäten heranziehen, um diese als Bedingung zu nutzen – also beispielsweise, um alle Bestellungen eines bestimmten Kunden zu ermitteln.

Damit kommen wir nun zu LINQ to Entities-Abfragen, die mehr als eine Entität betreffen. Dabei wollen wir gleich mit dem genannten Beispiel beginnen und alle Bestellungen des Kunden liefern, dessen Nachname **Minhorst** lautet. Da wir nun mit Bestellungen als Ergebnis arbeiten, definieren wir dazu eine entsprechende ObservableCollection samt öffentlicher Eigenschaft:

```
private ObservableCollection<Bestellung> bestellungen;
public ObservableCollection<Bestellung> Bestellungen {
    get { return bestellungen; }
    set { bestellungen = value; }
}
```

Der **case**-Zweig, der alle Bestellungen für den Kunden mit dem Wert **Minhorst** im Feld **Nachname** liefert, sieht nun so aus:

EDM: Backend ändern

Unsere aktuellen Beispieldatenbanken haben ein WPF/C#-Frontend, ein SQLite-Backend und das Entity Data Model als Zwischenschicht. Was geschieht nun, wenn wir einmal die Datenbank ändern wollen – etwa, weil wir neue Tabellen oder neue Felder in bestehenden Tabellen benötigen? Wie ist die genaue Vorgehensweise, die auch berücksichtigt, dass der Benutzer eine neue Version installieren möchte, ohne die bereits im vorhandenen Backend gespeicherten Daten zu verlieren? Wie dies gelingt, zeigt der vorliegende Artikel.

Bei der Entwicklung würden wir so vorgehen, dass wir das Datenmodell der SQLite-Datenbank ändern, indem wir die gewünschten Tabellen oder Felder hinzufügen. Dann wechseln wir zu Visual Studio und aktualisieren das Entity Data Model, gegebenenfalls müssen wir noch die Benutzeroberfläche beziehungsweise die Code behind-Klassen anpassen.

Aber was, wenn die Anwendung bereits beim Kunden im Produktiveinsatz ist? Dann würden wir die Entwicklung natürlich genau so betreiben, wie wir es oben beschrieben haben. Allerdings können wir natürlich nicht so einfach die neue Version des Datenbankbackends beim Kunden installieren, denn dieser hat ja bereits ein Backend mit gegebenenfalls neuen Daten, die wir dann überschreiben würden. Also was tun? Die neue Anwendung exklusive Backend können wir ja bei Kunden über die alte Version installieren, aber für das Backend müssen wir uns etwas einfallen lassen. Denn das neue Frontend wird auch spätestens beim Versuch, auf eines der neuen Felder per Entity Framework zuzugreifen, Probleme verursachen.

Allerdings gibt es ja die Möglichkeit, Datenmodelle per Code zu ändern. Wir können zum Beispiel neue Tabellen anlegen oder bestehende Tabellen entfernen, Felder zu Tabellen hinzufügen oder entfernen oder auch Einschränkungen ändern. Allerdings will dies gut geplant sein, denn vielleicht wollen wir das Datenmodell des Backends nicht nur einmal ändern, sondern im Laufe der Zeit je nach den geänderten Anforderungen auch öfter.

Versionsstand prüfen

In diesem Fall müssen wir in irgendeiner Form einen Versionsstand des Backends speichern, damit wir wissen, welche Aktualisierungen in Abhängigkeit vom Versionsstand durchzuführen sind. Da wir mit einem Datenbankbackend arbeiten, bietet sich dazu natürlich eine Tabelle an, die wir etwa **Versions** nennen. Die Tabelle braucht lediglich ein Feld namens **Version**, in das wir dann den Versionsstand der Tabelle eintragen.

Wir würden dann eine neue Version des Frontends installieren, diese starten und nach dem Start soll das Frontend dann prüfen, ob das Backend die dem Frontend entsprechende Version aufweist. Dazu soll das Frontend dann die Tabelle **Versions** öffnen und den Wert des Feldes **Version** prüfen. Wenn die Version kleiner als die geforderte Version ist, soll das Frontend die benötigten Änderungen am Backend durchführen und schließlich den Versionsstand in der Tabelle **Versions** aktualisieren.

Natürlich müssen wir auch einen umgekehrten Mechanismus vorsehen, der reagiert, wenn der Versionsstand des Backends aktueller ist als der des Frontends. Das kann etwa der Fall sein, wenn der Kunde das Frontend an mehr als einem Arbeitsplatz einsetzt und dieses nur an einem Arbeitsplatz auf den aktuellen Versionsstand bringt. Sollte dann ein Benutzer auf einem ande-

ren Arbeitsplatz ein veraltetes Frontend starten, sollte dieses mit einer entsprechenden Meldung darauf hinweisen, dass das Frontend aktualisiert werden soll.

Dazu gibt es natürlich noch weiterführende Techniken, bei denen das Frontend beim Start automatisch über das Internet prüft, ob es eine aktuellere Version des Frontends gibt, doch das soll nicht das Thema dieses Artikels sein.

Versionstabelle hinzufügen

Wir wollen die Schritte dieses Artikels anhand unserer Beispieldatenbank **Bestellverwaltung.db** auf Basis von SQLite erläutern. Diese enthält im aktuellen Zustand noch gar keine Tabelle namens **Versions**, sodass der erste Schritt sein wird, diese Tabelle dem Entwicklungsbackend hinzuzufügen und das Frontend um eine Funktion zu ergänzen, die beim Starten des Frontends prüft, ob die Tabelle **Versions** im Backend vorhanden ist oder nicht. Falls nicht, soll diese einfach ergänzt werden. Der Einfachheit halber finden Sie die Lösung in einer stark reduzierten Version der in den übrigen Artikeln verwendeten Anwendung, die nur das Fenster **MainWindow** enthält.

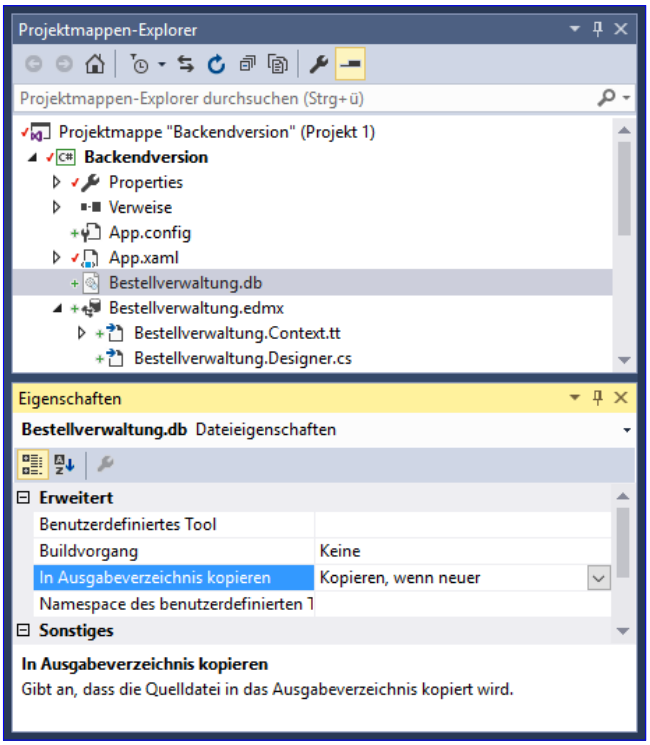


Bild 1: Eigenschaften der Datenbankdatei

Ort von Backendänderungen bei der Entwicklung

Wenn Sie, wie wir in diesem Artikel, mit einem Backend auf Dateibasis arbeiten (also mit SQLite und nicht etwa mit SQL Server), müssen Sie sich entscheiden, an welcher Stelle Sie die Änderungen durchführen. Wir führen eine Version des Backends direkt im Projektordner, also im gleichen Verzeichnis wie etwa die **.xaml**- und **.cs**-Dateien (später werden wir diese gegebenenfalls auch einmal in eigenen Ordnern strukturieren, aber aktuell reicht ein Verzeichnis dazu aus). Dieses fügen wir auch zum Projektmappen-Explorer hinzu. Dann stellen wir die Eigenschaft In **Ausgabeverzeichnis kopieren** für diesen Eintrag auf **Kopieren, wenn neuer** ein (siehe Bild 1). Die Datenbankdatei wird dann nur erneut in das Ausgabeverzeichnis kopiert, wenn die Ausgangsdatei aktualisiert wurde und somit ein neueres Änderungsdatum als die im Ausgabeverzeichnis enthaltene Datei enthält.

Hinzufügen der Tabelle Versions

Nun starten wir **SQLite Studio**, um die Tabelle **Versions** zur Datenbank hinzuzufügen. Hier wählen Sie den Menübefehl **Datenbank|Datenbank hinzufügen** und wählen im folgenden Dateiauswahl-Dialog die Datei **Bestellverwaltung.db** aus unserem Projektverzeichnis aus. Klicken Sie dann doppelt auf den Eintrag für diese Datenbank, sodass die enthaltenen Tabellen angezeigt werden. Klicken Sie mit der rechten Maustaste auf den Eintrag **Tabellen** und wählen Sie den Kontextmenü-Eintrag **Tabelle erstellen** aus. Geben Sie oben als Tabellenname **Versions** an und fügen Sie mit der Schaltfläche **Add Column** eine neue Spalte zur Tabelle hinzu. Dieses soll das Primärschlüsselfeld mit dem Spaltennamen **VersionID** und dem Datentyp **INTEGER** sowie automatischem Hinzufügen der Feldwerte sein. Für das zweite Feld legen Sie als Spaltenname **Version** fest und geben als Datentyp **String** mit der Länge **50** an (sicher ist sicher). Danach sieht die Definition der Tabelle wie in Bild 2 aus

und Sie können mit einem Klick auf die Schaltfläche **Commit structure changes** die Änderungen in die Datei schreiben.

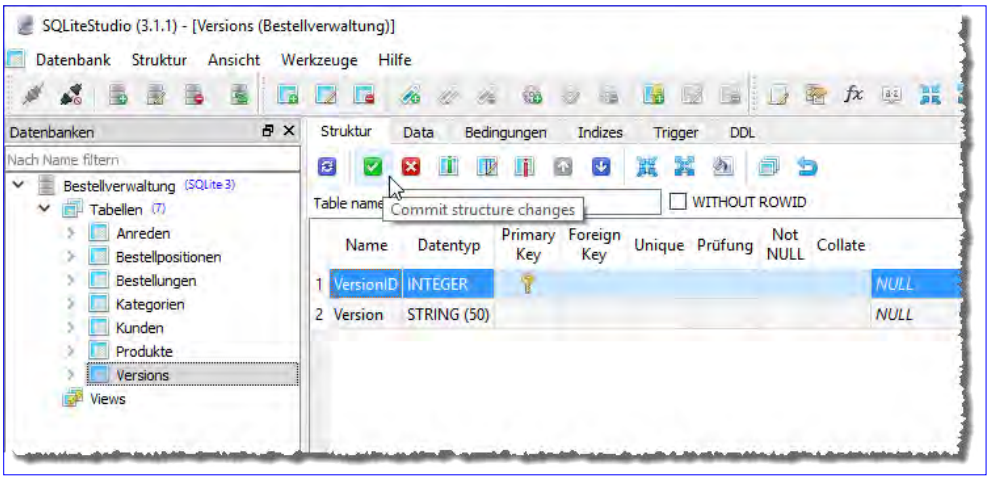


Bild 2: Anlegen der Tabelle **Versions**

Es erscheint dann der Dialog aus Bild 3 mit dem SQL-Code der auszuführenden Abfrage. Das ist recht praktisch, denn so können Sie den Code direkt kopieren und diesen später hervorholen, wenn Sie das Anlegen dieser Tabelle per C# programmieren wollen.

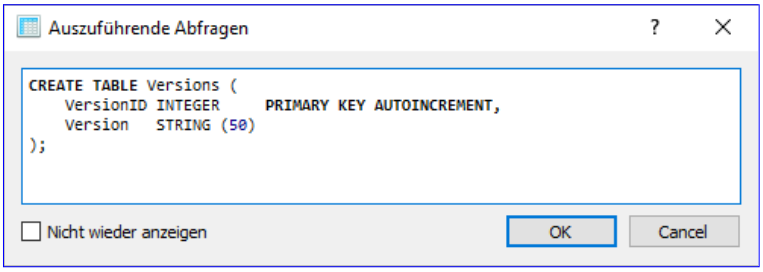


Bild 3: SQL-Code zum Erstellen der neuen Tabelle

Das waren schon alle Schritte, die wir von hier aus erledigen wollen.

Geändertes Datenmodell in Entity Data Model übernehmen

Danach müssen wir das Entity Data Model an das neue Datenmodell anpassen. Dazu öffnen Sie dieses per Doppelklick auf de Eintrag **Bestellverwaltung.edmx** im Projektmappen-Explorer. Dies öffnet die Übersicht, die per Klick mit der rechten Maustaste den Kontextmenü-Befehl **Modell aus der Datenbank aktualisieren...** offenbart (siehe Bild 4). Es erscheint dann nach einigen Sekunden der Update-Assistent und bittet um die Auswahl der Datenverbindung (eventuell erkennt er auch die korrekte Verbindung und zeigt direkt den folgenden

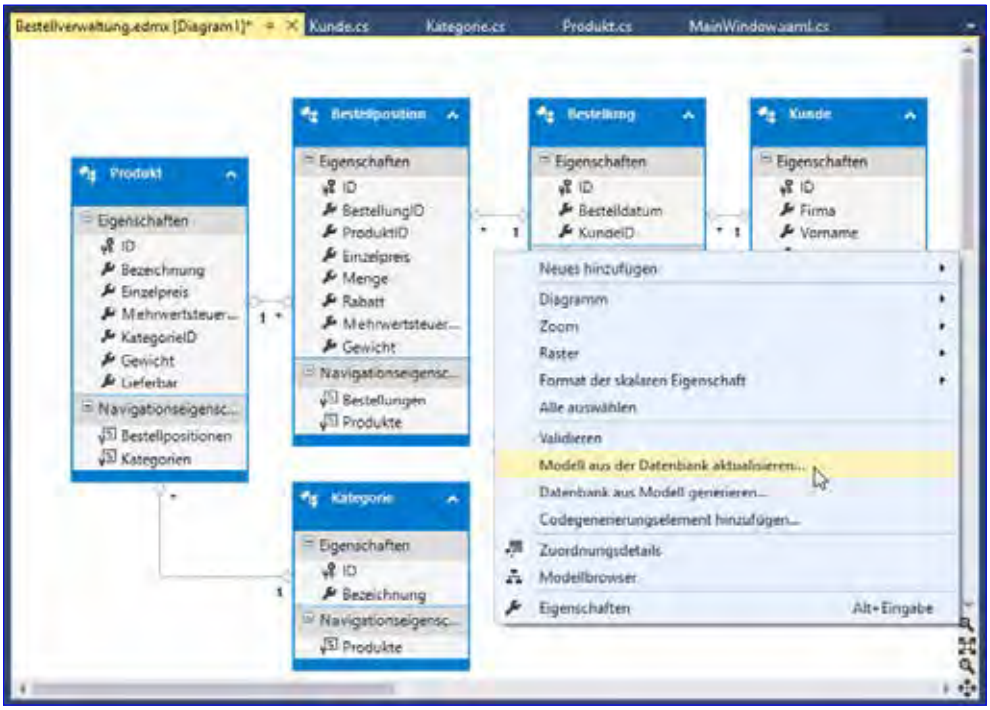


Bild 4: Aktualisieren des Modells aus der Datenbank

Dialog an). Die vorhandene Einstellung können Sie in der Regel beibehalten und mit einem Klick auf die Schaltfläche **Weiter** den nächsten Schritt aufrufen.

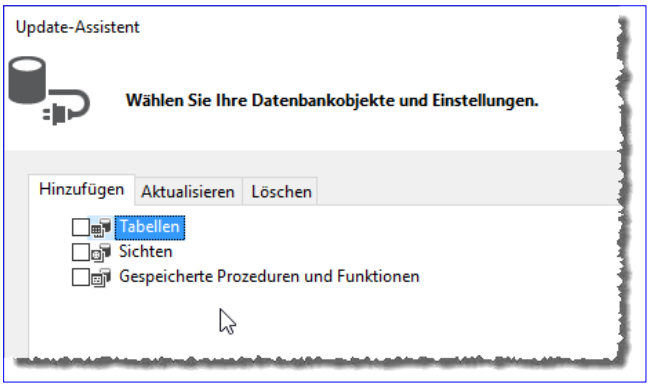


Bild 5: Es stehen keine Objekte zur Aktualisierung zur Verfügung.

Nun kann es zu folgendem kommen: Es erscheint der Schritt mit der Überschrift **Wählen Sie Ihre Datenbankobjekte und Einstellungen**., aber es lässt sich keine der Optionen **Tabellen**, **Sichten** oder **Gespeicherte Prozeduren und Funktionen** aktivieren (siehe Bild 5). Der Grund für diesen Zustand ist vermutlich, dass es tatsächlich keine zu aktualisierenden Objekte gibt. Das kann zwei Ursachen haben:

- Sie haben die Änderungen nicht in die Datenbank übernommen. Prüfen Sie, ob zum Beispiel die neue Tabelle Versions in der Liste der Tabellen in **SQLite Studio** angezeigt wird.

- Sie haben die Änderungen an einer anderen Datenbank vorgenommen, als an der, die Sie nun zum Übertragen der Änderungen in das Entity Data Model verwenden wollen.

Wenn Sie den ersten Fall ausschließen können, sollten Sie im Update-Assistenten für das Entity Data Model nochmal zum ersten Schritt zurückkehren. Hier stellen Sie dann unter Umständen fest, dass Sie mehrere Verbindungen über das Kombinationsfeld auswählen können und vielleicht die falsche Datenbank ausgewählt haben. Hier sollten Sie dann sichergehen, dass Sie die richtige gewählt haben – beispielsweise, indem Sie unten unter Verbindungszeichenfolge prüfen, ob dort auch tatsächlich die Datenbankdatei im richtigen Verzeichnis angegeben ist (siehe Bild 6).

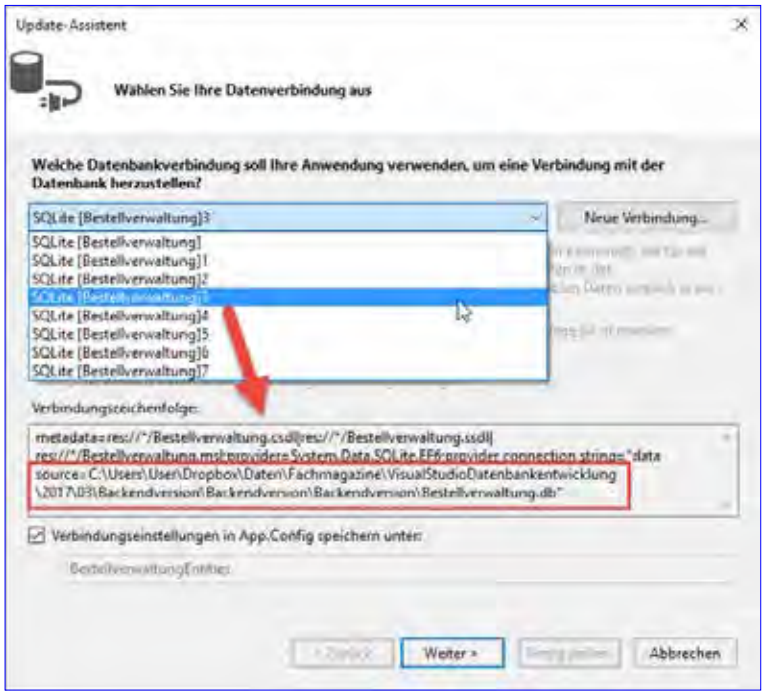


Bild 6: Prüfen der korrekten Datenverbindung anhand der Verbindungszeichenfolge

Achtung: Wenn Sie mit den Beispieldateien aus dem Download arbeiten, müssen Sie auf jeden Fall eine neue Verbindung anlegen und die Quelldatenbank neu auswählen – außer, Sie verwenden zufällig genau die gleiche Verzeichnisstruktur wie wir.

Wenn Sie danach einen Schritt weitergehen, erscheint die Tabelle **Versions** auch in der Liste der zur Aktualisierung bereitstehenden Tabellen und Sie können diese nun auswählen (siehe Bild 7). Klicken Sie dann auf **Fertigstellen**, erscheint schon kurz danach die Tabelle **Versions** im Entity Data Model.

Sie können somit auch über das Entity Data Model auf die Tabelle und ihre Inhalte zugreifen.

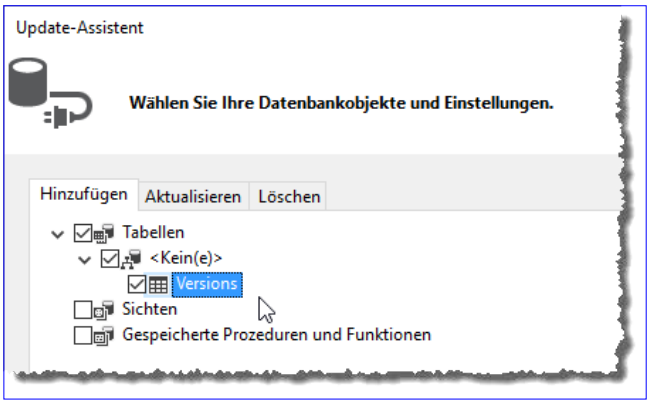


Bild 7: Einbinden der Tabelle **Versions** in das Entity Data Model

Code zum Prüfen der Version des Backends

Nun wollen wir Code zur Anwendung hinzufügen, der beim Öffnen prüft, ob die richtige Version des Backends vorliegt. Dabei unterscheiden wir zwei Fälle:

- Das Backend enthält noch gar keine Tabelle **Versions**. Dann muss diese zunächst angelegt werden. Danach sollen alle Updates des Backends bis zum aktuellen Versionsstand ausgeführt werden.
- Die Tabelle **Versions** liegt bereits vor. Dann prüfen wir den Versionsstand des Backends und führen alle Updates des Backends bis zum aktuellen Versionsstand aus.

Ort des Update-Codes

Bevor wir mit der Programmierung beginnen, müssen wir uns überlegen, wohin wir den Code schreiben, der die Version prüfen und eventuell notwendige Updates durchführt. Im Artikel **Entity Framework: SQLite verknüpfen** haben wir ja bereits eine Funktion programmiert, die beim Anzeigen von **MainWindow.xaml** ausgeführt wird und prüft, ob das Backend sich überhaupt an Ort und Stelle befindet und den Benutzer gegebenenfalls bittet, dieses auszuwählen. Dies ist ein geeigneter Ort, um auch die nachfolgend vorgestellten Methoden aufzurufen.

Prüfen, ob die Tabelle Versions vorhanden ist

Um zu prüfen, ob die Tabelle vorliegt, versuchen wir einfach, den Wert des Feldes **Version** zu lesen. Dabei kann es drei Fälle geben:

- Die Tabelle ist nicht vorhanden, was einen Fehler auslöst.

AUSGABE 05/2017

DATENBANK ENTWICKLER

MAGAZIN FÜR DIE DATENBANKENTWICKLUNG MIT VISUAL STUDIO FÜR DESKTOP, WEB UND CO.

TOP-THEMEN:

NEWS AND TOOLS	LINQPad: LINQ, C# und VB einfach ausprobieren	SEITE 1
USERINTERFACE	Drag and Drop-Grundlagen	SEITE 1
VB-BASICS	Von C# zu VB: Basics, Variablen, Operatoren	SEITE 21
INTERAKTIV	Excel-Export mit Spreadsheet Light	SEITE 47
LÖSUNGEN	Bestellverwaltung á la Visual Basic	SEITE 54

LINQPad: LINQ, C# und VB einfach ausprobieren

Es ist doch etwas nervig, wenn man mal eben ein paar LINQ-Abfragen testen möchte oder auch C#- oder VB-Code. Immer wieder den Code anpassen, kompilieren, starten, feststellen, dass es noch nicht wie gewünscht klappt und wieder von vorn. Unter Access/VBA ging das irgendwie einfacher: Prozedur schreiben, F5 drücken, fertig. Das Tool LINQPad erleichtert die Sache stark: Damit können Sie prima üben, ohne zum Testen immer gleich die Anwendung kompilieren zu müssen.

LINQPad

LINQPad ist eine Anwendung, die in einer abgespeckten Version kostenlos ist und nach der Installation gegen Entgelt mit einigen interessanten Features wie IntelliSense und mehr aufgerüstet werden kann. Die Bezeichnung von LINQPad heißt nicht umsonst **The .NET Programmer's Playground**: Hier können Sie viele Ausdrücke in den Sprachen C#, Visual Basic oder auch F# ausprobieren. Uns interessiert natürlich vor allem der Zugriff auf Datenbanken, denn wir wollen ja im Rahmen dieses Artikels weitere Methoden von LINQ to Entities am Beispiel unserer Datenbank **Bestellverwaltung** ausprobieren.

Den Download von LINQPad finden Sie unter der Internetadresse **www.linqpad.net**.

Nachdem Download und der Installation können Sie gleich loslegen und LINQPad starten. Die Benutzeroberfläche besteht aus drei Elementen: Links oben finden Sie einen Bereich, mit dem Sie die Verbindungen zu Ihren Datenquellen verwalten können. Links unten können Sie in bereits vorhandenen Beispielen suchen oder in dem für LINQPad definierten Bereich des Dateisystems Ihre eigenen Beispiele verwalten. Rechts finden Sie ein Register-Steuerelement, das ein oder mehrere Registerkarten mit Ihren selbst erstellen oder geladenen Skripten anbietet. Oben sind zwei wichtige Kombinationsfelder: **Language** legt nicht nur fest, mit welcher Programmiersprache Sie arbeiten, sondern auch, welche Art von Anweisungen Sie nutzen möchten (siehe Bild 1). Für unsere Zwecke, in denen wir zunächst nur LINQ-Ausdrücke testen wollen, reicht die Auswahl des Eintrags **C# Statement(s)** aus. Wenn Sie die Ausdrücke unter VB testen wollen, nutzen Sie den entsprechenden Eintrag.

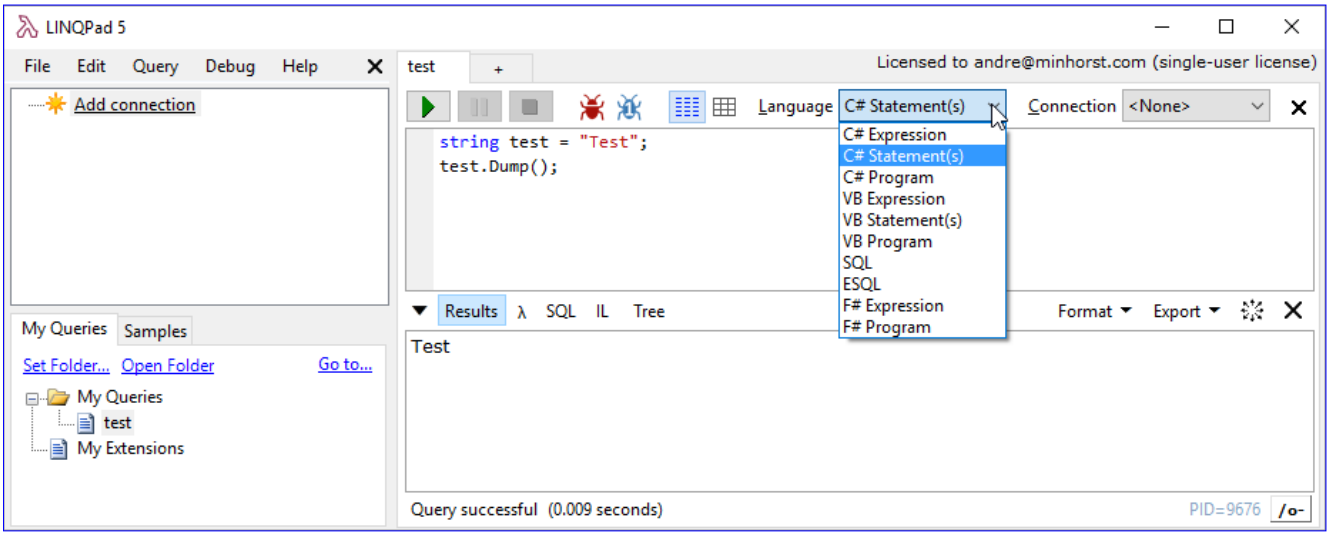


Bild 1: Ein erster Test mit LINQPad

Datenquelle wählen

Neben der Sprache benötigen wir natürlich eine Datenquelle. Um diese einzustellen, klicken Sie auf den Link **Add connection** im linken, oberen Bereich.

Dies öffnet den Dialog **Choose Data Context** (siehe Bild 2). Hier finden Sie zwei grundlegende Möglichkeiten:

- **Build data context automatically**: Erwartet die Angabe der Datenquelle, in unserem Fall zum Beispiel einer SQLite-Datenbankdatei, und erstellt die Objekte automatisch auf Basis der enthaltenen Tabellen und Beziehungen.
- **Use a typed data context from your own assembly**: Verwendet den in den Konfigurationsdateien definierten Datenbank-Kontext als Grundlage für die Entitäten. Dies hat mit SQLite in unseren Versuchen allerdings nicht geklappt.

Die erste Option bietet leider keinen passenden Treiber für unsere SQLite-Datenbank **Bestellverwaltung.db**. Allerdings finden wir unten im Dialog noch eine Schaltfläche mit der Beschriftung **Browse...**, mit der wir den Dialog aus Bild 3 öffnen.

Hier finden wir dann gleich ganz oben den Eintrag, der uns die Treiber unter anderem für SQLite nachreicht. Mit einem Klick auf **Download & Enable Driver** fügen wir den fehlenden Treiber hinzu.

Dieser erscheint dann nach dem Schließen des Dialogs auch in der oberen Liste des Dialog **Choose Data Context** (siehe Bild 4).

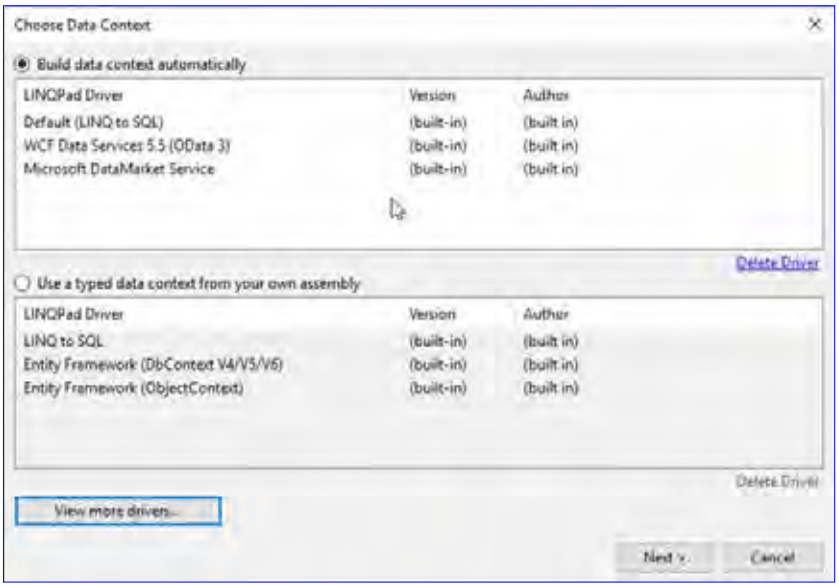


Bild 2: Auswählen des Datenbank-Kontexts

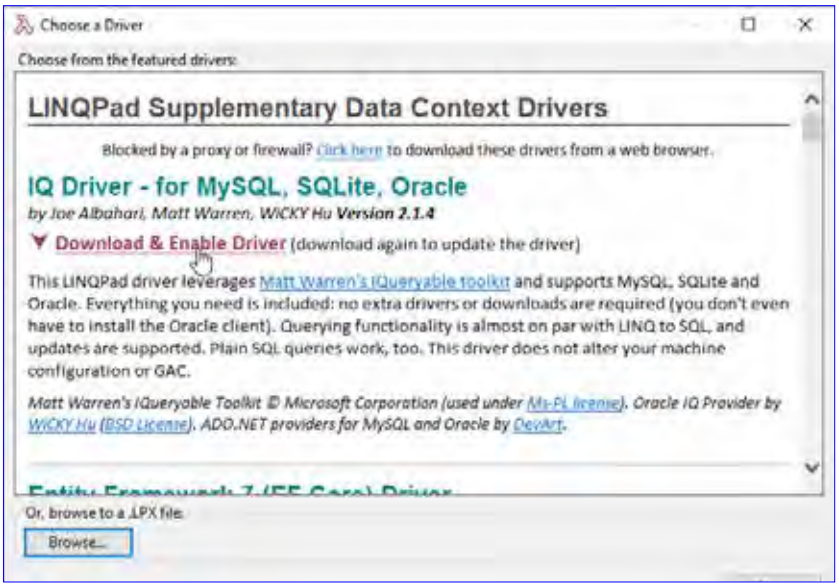


Bild 3: Herunterladen des Treibers für MySQL, SQLite und Oracle

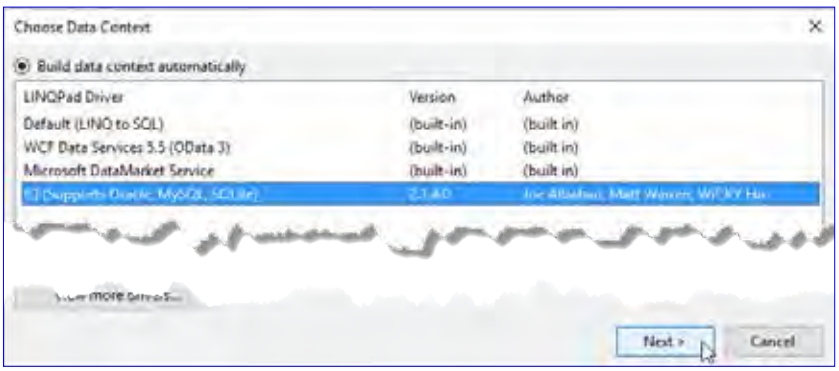


Bild 4: Auswählen des SQLite-Treibers

Property-Elemente am Beispiel Content

Unter Access konnten Sie beispielsweise in einer Schaltfläche lediglich einen Text als Inhalt unterbringen. In neueren Versionen kamen dann noch einige Features zur Gestaltung von Rand, Hintergrund, Bildern et cetera hinzu. Unter WPF sieht das ganz anders aus. Eine Schaltfläche bringt zwar mit der Content-Eigenschaft die Möglichkeit mit, einfache Texte einzugeben und liefert auch noch einige weitere Attribute, mit denen sich andere gängige Eigenschaften wie Rahmen, Hintergrund und so weiter einstellen lassen. Aber die Content-Eigenschaft nimmt nicht nur reine Texte, sondern nahezu beliebige Inhalte entgegen und zeigt diese innerhalb des Buttons an. Der Button ist dabei übrigens nur ein Beispiel – es gibt noch weitere Steuerelemente, welche die Content-Eigenschaft anbieten.

Einfache Texte

Wenn Sie wie beispielsweise unter Access einfach nur einfache Texte in einer Schaltfläche anzeigen möchten, weisen Sie diese einfach der Eigenschaft **Text** zu. Unter WPF gibt es dafür das Attribut **Content**, das wir im folgenden Beispiel mit dem Text **Beispielbutton** füllen (Ergebnis siehe Bild 1):

Bild 1: Button mit einfachem Text

```
<Button Height="25" Width="120" VerticalAlignment="Top" HorizontalAlignment="Left" Margin="10"
Content="Beispielbutton"></Button>
```

Content mit anderem Objekt füllen

Wenn Sie nun nicht nur einen einfachen Text als **Content** verwenden möchten, sondern beispielsweise ein anderes Objekt wie einen Text mit einem Bild, scheint dies mit dem **Content**-Attribut allein nicht möglich. Unter C# könnten Sie das aber beispielsweise wie folgt lösen:

```
public partial class MainWindow : Window {
    public MainWindow() {
        InitializeComponent();
        Button btn = new Button();
        btn.Width = 150;
        btn.Height = 32;
        StackPanel stp = new StackPanel();
        Image img = new Image();
        img.Source = new BitmapImage(new Uri(@"/ContentProperty;component/images/close.png", UriKind.Relative));
        img.Width = 24;
        img.Height = 24;
```

```
        img.Margin = (Thickness)System.ComponentModel.TypeDescriptor.GetConverter(typeof(Thickness)).
            ConvertFromInvariantString("5,0,5,0");
        TextBlock txt = new TextBlock();
        txt.Text = "Button mit Bild";
        txt.VerticalAlignment = VerticalAlignment.Center;
        stp.Children.Add(img);
        stp.Children.Add(txt);
        stp.Orientation = Orientation.Horizontal;
        btn.Content = stp;
        btn.HorizontalContentAlignment = HorizontalAlignment.Left;
        MyGrid.Children.Add(btn);
    }
}
```

Zusätzlich müssen Sie das **Grid**-Element im **.xaml**-Code noch entsprechend benennen, damit wir über die Bezeichnung **MyGrid** darauf zugreifen können:

```
<Grid x:Name="MyGrid">
```

Die Methode **MainWindow** erzeugt nun ein neues **Button**-Element, dem es zunächst eine Breite und eine Höhe zuweist. Dann erstellt sie ein neues **StackPanel**-Element, das die beiden enthaltenen Elemente, also das Bild und den Text nebeneinander anordnen soll. Das Bild bringen wir zunächst in der Variablen **img** unter und weisen seiner **source**-Eigenschaft über **new BitmapImage** ein im Projekt gespeichertes Bild zu. Danach stellt die Methode Höhe und Breite des Bildes ein sowie die horizontale Ausrichtung.

Bild 2: Per Code erstellte Schaltfläche mit Bild und Text

Den Abstand des **image**-Elements zum linken Rand des StackPanels und zum rechts daneben liegenden **TextBlock**-Element, das gleich noch erstellt wird, wollen wir mit dem **Margin**-Attribut einstellen. Was unter **.xaml** schnell per Zuweisung gelingt, erfordert hier den Einsatz einer Struktur des Typs **Thickness**, der wir für die Attribut **Left** und **Right** die entsprechenden Zahlenwerte zuweisen. Nun erstellen wir noch den **TextBlock**, der schließlich den gewünschten Text aufnimmt. Der Text soll vertikal zentriert eingeblendet werden. Nun können wir über die Methode **Add** der **Children**-Auflistung des **StackPanel**-Elements den Button und den TextBlock hinzufügen. Das StackPanel weisen wir dann der Eigenschaft **Content** des **Button**-Elements zu, das wir noch linkszentriert ausrichten, bevor wir das **Button**-Element noch im Grid einsetzen. Das Ergebnis sieht dann wie in Bild 2 aus.

Komplexer Content per .xaml

Wie nun wollen wir diese aufwendig per C# zusammengestellten Elemente per **.xaml** zu einem Window hinzufügen, wo wir doch nur das **Content**-Attribut zur Verfügung haben? Der Clou ist, dass es für **Button**- und andere Elemente noch eine weitere Möglichkeit gibt, den Content anzugeben, nämlich einfach in Form von Unterelementen. Das bedeutet, dass wir das **Content**-

Drag and Drop-Grundlagen

Unter Access fehlen einige Features, die in anderen Programmiersprachen und Entwicklungsumgebungen zum guten Ton gehören. Eines davon ist die Drag and Drop-Funktionalität, die sich nur aufwendig abbilden ließ – und auch nur mit bestimmten ActiveX-Steuerelementen. Die eingebauten Steuerelemente wie Textfelder oder Listenfelder ließen leider kein natives Drag and Drop zu. Unter WPF und den .NET-Programmiersprachen sieht das ganz anders aus. Dieser Artikel liefert Grundlagen zu Drag and Drop.

Wozu braucht man aber Drag and Drop? In der Tat lassen sich die meisten Tätigkeiten, die Sie damit ausführen können, auch auf andere Weise realisieren. Allerdings ist Drag and Drop und somit das einfache Ziehen von Elementen der Benutzeroberfläche mit der Maus doch ein sehr intuitiver Weg, um Aktionen durchzuführen. Ein Beispiel sind die beiden Listenfelder aus dem Artikel **m:n-Beziehung mit Listenfeld** (siehe Bild 1). Im Artikel haben wir Schaltflächen bereitgestellt, mit denen ein oder alle Artikel von einem Listenfeld zum anderen übertragen werden können. Nun wollen wir uns weiter vortasten und Drag and Drop-Funktionalität zu diesem Fenster hinzufügen. Bevor wir uns das im Artikel **Drag And Drog mit ListBox**-Elementen ansehen, schauen wir uns einige grundlegende Beispielen an.

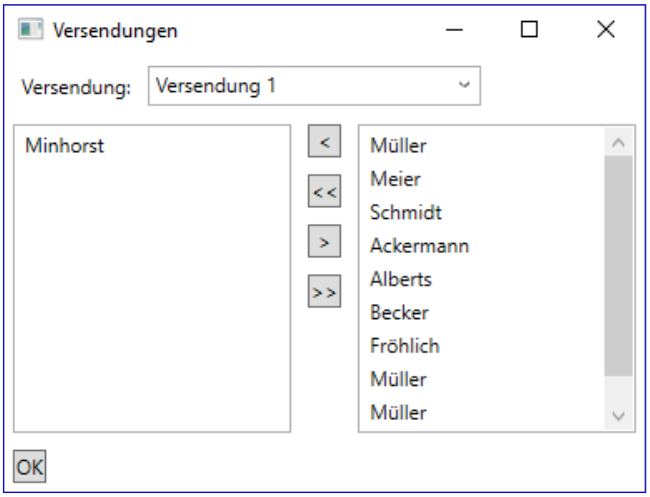


Bild 1: Für das Hin- und Herziehen von Einträgen zweier Listenfelder wäre Drag and Drop eine interessante Alternative.

Was ist Drag and Drop?

Drag and Drop nennen wir den Vorgang, bei dem der Benutzer ein Element der Benutzeroberfläche mit der linken Maustaste anklickt und diese dabei gedrückt hält. Dann wird die Maus mit dem Element zum Zielelement bewegt und das zu bewegende Element dort durch Loslassen der linken Maustaste an das Zielelement übergeben.

Dabei müssen Start- und Zielelement gar nicht unbedingt zwei verschiedene Elemente sein wie beim Beispiel mit den Listenfeldern. Es kann auch nur ein einziges Listenfeld sein, dessen Einträge Sie per Drag and Drop umsortieren wollen. Oder Sie bewegen die Elemente in einem **TreeView**-Steuerelement von einem übergeordneten Element zum nächsten.

Dazu benötigen wir ein paar Informationen. Zunächst müssen wir für das abgebende Element festlegen, dass es das Ziehen der enthaltenen Elemente mit der Maus überhaupt erlaubt. Dann wollen wir uns irgendwie merken, welches Element wir mit der Maus angepackt haben, um es zu verschieben (oder auch zu kopieren). Schließlich soll das aufnehmende Element mit der Möglichkeit zum Aufnehmen des zu übergebenden Elements ausgestattet werden, also als Drag and Drop-Ziel verfügbar gemacht werden. Da man unter Windows eine ganze Menge Dinge durch die Gegend ziehen kann – zum Beispiel von Windows Explorer auf verschiedene andere Ziele – müssen Sie dem Zielelement auch noch mitteilen, auf welche zu übergebenden Elemente es überhaupt reagieren soll, indem es das Symbol als mögliches Zielelement einblendet. Schließlich benötigen wir verschiedene

Ereignisse, für die wir passende Methoden implementieren, um das Verhalten von Drag and Drop-Quelle und -Ziel festzulegen.

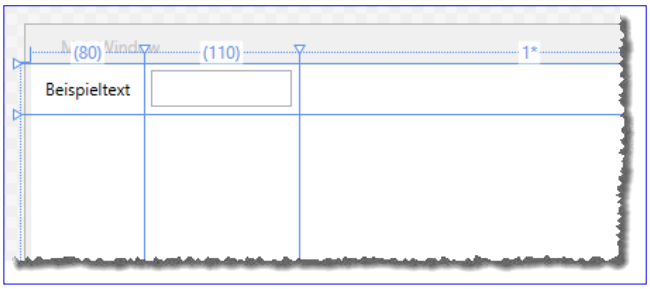


Bild 2: Label und TextBox als erstes Drag and Drop-Beispiel

Drag and Drop ist auch eine Alternative zum Kopieren/Ausschneiden und Einfügen. Der wesentliche Unterschied neben den dazu notwendigen Techniken (Ziehen mit der Maus bei Drag and Drop auf der einen, Tastenkombinationen wie **Strg + X**, **V** oder **C** und Kontextmenü-Einträge auf der anderen Seite) ist der Ort, an dem wir uns merken, welches Element von A nach B bewegt werden sollen. Beim Kopieren/Ausschneiden und Einfügen ist dies die Zwischenablage. Bei Drag and Drop benötigen wir eine andere Möglichkeit, in diesem Fall ein Element des Typs **DataObject**.

Einfaches Beispiel: Bezeichnungsfeld in Textfeld ziehen

Wir schauen uns zu Beginn ein einfaches Beispiel an, bei dem wir die Beschriftung eines Bezeichnungsfeldes einfach in ein Textfeld ziehen wollen. Dazu legen wir entsprechende Steuerelemente in einem neuen Fenster namens **MainWindow.xaml** eines neuen Projekts an (s. Bild 2). Die beiden Steuerelemente definieren wir wie wie folgt:

```
<TextBox x:Name="txtDrop" Grid.Column="1" Grid.Row="1" Margin="5" Drop="txtDrop_Drop" Width="100" />
<Label x:Name="lblDrag" Content="Beispieltext" Grid.Column="0" Grid.Row="1" Margin="5" MouseDown="lblDrag_MouseDown" />
```

Unter C# sieht die Ereignismethode für das Ereignis **MouseDown** des Bezeichnungsfeldes wie folgt aus:

```
//C#
private void lblDrag_MouseDown(object sender, MouseButtonEventArgs e) {
    DragDrop.DoDragDrop(lblDrag, lblDrag.Content, DragDropEffects.Copy);
}
```

Wir benötigen also nur eine einzige Zeile, um den Vorgang zu starten! Dabei rufen wir die Methode **DoDragDrop** der Klasse **DragDrop** auf. Diese erwartet drei Parameter:

- **dragSource**: Quelle des Drag and Drop-Vorgangs, also beispielsweise ein Steuerelement wie in diesem Fall das Bezeichnungsfeld
- **data**: Die per Drag and Drop bewegten Daten werden mit dem Parameter **data** in einen Zwischenspeicher ähnlich der Zwischenablage gegeben und können dort beim Drop wieder abgerufen werden.
- **allowedEffects**: Effekte sind verschiedene Darstellungsarten des Mauszeigers.

Im vorliegenden Fall übergeben wir für den ersten Parameter einen Verweis auf das Bezeichnungsfeld **lblLabel**. Der zweite Parameter nimmt den Inhalt der Eigenschaft **Content** des Bezeichnungsfeldes auf, hier eine einfache Beschriftung. Da es sich

hierbei um ein reines Kopieren von Daten handelt, nämlich das Kopieren der Beschriftung des Bezeichnungsfeldes in das Textfeld, soll nur das Symbol für das Kopieren mit der Maus angezeigt werden (siehe Bild 3).

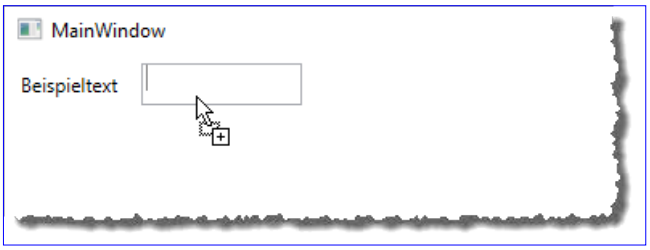


Bild 3: Veränderter Mauszeiger beim Drag and Drop

Damit kommen wir Fallenlassen des zu kopierenden Inhalts über dem Zielobjekt. Hier verwenden wir nun das Ereignis Drop des Textfeldes, das wir wie folgt implementieren:

```
//C#
private void txtDrop_Drop(object sender, DragEventArgs e) {
    string str;
    str = (string)e.Data.GetData(DataFormats.StringFormat);
    txtDrop.Text = str;
}
```

Die Methode liefert die gewohnten beiden Parameter, wobei der zweite den Typ **DragEventArgs** hat (mehr dazu weiter unten). In diesem Fall nutzen wir das Objekt **data** der mit dem Parameter e übergebenen **DragEventArgs**-Klasse, dessen Methode **GetData** mit dem Parameter **DataFormats.StringFormat** den gespeicherten Text zurückgibt. Diesen speichern wir in der String-Variablen **str** und weisen diese dann der **Text**-Eigenschaft des Textfeldes zu. Das Ergebnis sieht schließlich wie in Bild 4 aus.

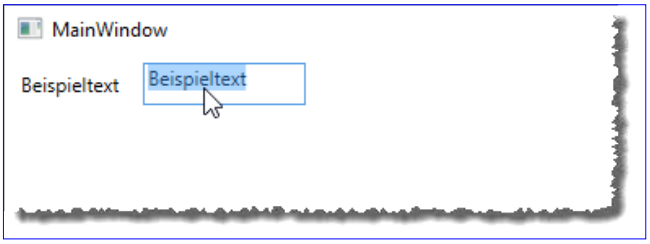


Bild 4: Erfolgreicher Drag and Drop-Vorgang

Zusammengefasst haben wir für den Start von Drag and Drop das Ereignis **MouseDown** genutzt, dort den Drag and Drop-Vorgang gestartet und dadurch auch das Ereignis **Drop** aktiviert. Ohne den Aufruf von **DoDragDrop** können Sie nämlich mit der Maus ziehen und fallenlassen, was sie möchten – das **Drop**-Ereignis wird nicht ausgelöst.

Verfeinerung: Maustaste prüfen

Im aktuellen Zustand können Sie auch mit der rechten Maustaste den Drag and Drop-Vorgang ausführen. Das liegt daran, dass wir in der Methode **lblDrag_MouseOver** nicht geprüft haben, welche Maustaste gerade gedrückt wurde. Dies holen wir nun nach und passen die Methode wie folgt an:

```
private void lblDrag_MouseDown(object sender, MouseButtonEventArgs e) {
    if (e.LeftButton == MouseButtonState.Pressed) {
        DragDrop.DoDragDrop(lblDrag, lblDrag.Content, DragDropEffects.Copy);
    }
}
```

Hier prüfen wir vorab, ob die linke Maustaste gedrückt ist. Damit wird das Drag and Drop mit gedrückter rechter Maustaste unterbunden.

Drag and Drop aus anderen Anwendungen

Wenn Sie einen Text aus einer anderen Anwendung in ein Textfeld einer WPF-Anwendung ziehen wollen, klappt das mit der gegebenen Konfiguration ebenfalls. Sie können nun beispielsweise Text aus einem Text-Editor oder aus Word markieren und diesen dann in das Textfeld ziehen. Aber: Dazu wären die bisherigen Schritte gar nicht nötig gewesen – Sie können auch einfach so den Text aus einer externen Anwendung in das Textfeld ziehen. Dazu müssen Sie noch nicht einmal ein Attribut des Textfeldes anpassen. Interessant ist noch die Frage: Feuert denn das Drop-Ereignis des Textfeldes, wenn Text aus einer externen Anwendung abgelegt wird? Die Antwort lautet nein.

Drag and Drop in andere Anwendungen

Bei Texten gelingt das Drag and Drop aus einem Textfeld in eine andere Anwendung, welche als Drop-Ziel ausgelegt ist wie etwa Microsoft Word, ohne weiteres Zutun.

Auch wenn man einen Drag and Drop-Vorgang in der WPF-Anwendung so beginnt, dass der zu verschiebende Inhalt in die Drag and Drop-Zwischenablage übertragen wird, können Sie den Inhalt etwa in eine Textverarbeitung übertragen. Auf diese Weise können Sie etwa den Text der Schaltfläche aus dem ersten Beispiel in eine Textverarbeitung statt in das Textfeld ziehen.

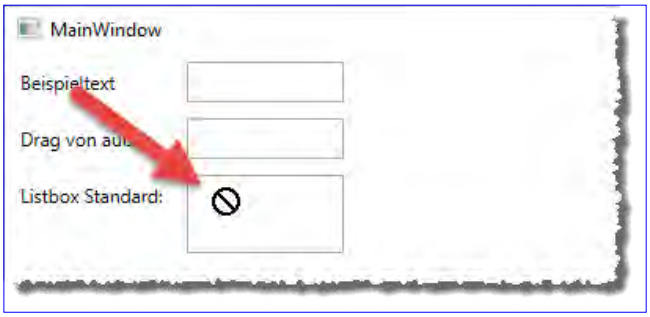


Bild 5: Drag and Drop auf eine Standard-ListBox

Drag and Drop in eine ListBox

Textfelder sind also immer für Drag and Drop empfänglich, wenn es sich um Daten im passenden Typ handelt, also um Texte. Was aber, wenn wir einen Text auf ein anderes Steuerelement droppen – beispielsweise ein **ListBox**-Steuerelement? Dazu fügen wir unserem Beispielformular eine ListBox hinzu und nennen diese **lstDrop**:

```
<ListBox x:Name="lstDropStandard" Grid.Column="1" Grid.Row="3" Margin="5" Width="100" Height="50" />
```

Wenn wir den Text des Bezeichnungsfeldes **lblDrag** auf das **ListBox**-Element ziehen, wird der Mauszeiger in ein Symbol umgewandelt, dass deutlich macht, dass das **ListBox**-Element kein valides Ziel für die Drag and Drop-Anweisung ist (siehe Bild 5). Also stellen wir nun das Attribut **AllowDrop** auf **True** ein:

```
<ListBox x:Name="lstDropStandard" ... AllowDrop="True" />
```

Damit erscheint schon einmal das richtige Symbol, wenn Sie das **ListBox**-Element beim Drag and Drop mit der Maus überfahren. Beim Fallenlassen geschieht jedoch nichts. Kein Wunder: Ein **ListBox**-Element nimmt seine Inhalte ja auch über ganz andere Wege entgegen – zum Beispiel über die **Add**-Methode der **Items**-Auflistung. Diese implementieren wir nun für die **Drop**-Methode des **ListBox**-Element, das wir dazu als neues Element wie folgt erweitern:

Von C# zu VB: Basics, Variablen, Operatoren

Wer bisher mit C# gearbeitet hat und zu VB wechseln möchte, sieht sich bei der Entwicklung von WPF-Anwendungen einigen Änderungen gegenüber. Dieser Artikel fasst die wichtigsten Elemente der Sprache Visual Basic 2015 für den Entwickler in Bezug auf die bisher in diesem Magazin unter C# durchgeführten Programmierungen zusammen. Zum Experimentieren mit den Beispielen nutzen wir das Tool LINQPad 5. Außerdem schauen wir uns in diesem Artikel die Grundlagen der Sprache an, die wir im Artikel »Von VBA zu C#: Erste Anwendung und Variablen« betrachtet haben.

WPF-Fenster: Wo ist der Code im Code behind-Modul?

Wer zuvor mit C# gearbeitet hat, weiß, dass ein Code behind-Modul eines WPF-Fensters bereits einigen Code enthält. Wenn Sie hingegen ein VB/WPF-Projekt erstellen und sich das Code behind-Fenster ansehen, erblicken Sie eine ziemliche Leere.

Dies ist der vollständige Inhalt des Moduls, hier am Beispiel von **MainWindow.xaml.vb**:

```
Class MainWindow
End Class
```

Konstruktor hinzufügen

Den liebgewonnenen Konstruktor, also die Methode, die automatisch beim Initialisieren des Fensters ausgelöst wird und die Sie auch so anpassen können, dass Sie beim Aufruf benutzerdefinierte Parameter übergeben können, fehlt gänzlich. Wie also wollen Sie beim Aktionen programmieren, die beim Initialisieren des Fensters durchgeführt werden geschweige denn eigene Parameter definieren?

Das geht einfacher, als Sie denken – Sie fügen einfach die Zeile **Public Sub New()** hinzu und erhalten dann die wie folgt automatisch ergänzte Methode:

```
Public Sub New()
        ' Dieser Aufruf ist für den Designer erforderlich.
        InitializeComponent()
        ' Fügen Sie Initialisierungen nach dem InitializeComponent()-Aufruf hinzu.
End Sub
```

Hier können Sie dann eigene, beim Initialisieren auszuführende Anweisungen eintragen. Außerdem können Sie die **New()**-Methode natürlich auch um benutzerdefinierte Parameter erweitern.

Groß- und Kleinschreibung

Was als Nächstes auffällt, ist die Großschreibung von Schlüsselwörtern. Dies ist einer der Unterschiede zwischen C# und Visual Basic 2015.

Case Sensitive oder nicht?

Case Sensitive bedeutet, dass die Groß- und Kleinschreibung einen Unterschied macht – **a** ist also ungleich **A**. Das heißt auch, dass Sie unter C# beispielsweise die folgenden beiden **String**-Variablen deklarieren können:

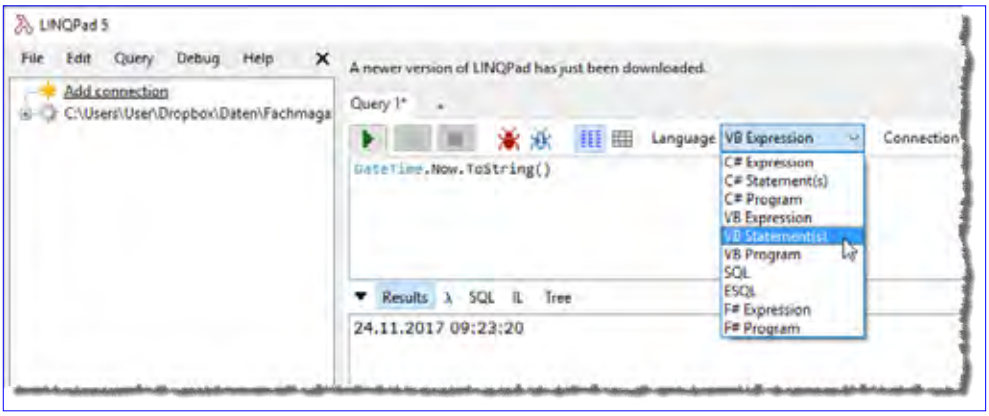


Bild 1: Auswahl der verschiedenen Eingabemöglichkeiten

```
string a;
string A;
```

Unter Visual Basic erhalten Sie für die zweite Zeile einen Fehler:

```
Dim a As String
Dim A As String
```

Variablen und Typen/Klassen können jedoch in beiden Sprachen mit unterschiedlicher Groß-/Kleinschreibung genutzt werden, also etwa wie folgt, wo wir **Test1** als Klassennamen und **test1** als Objektname auf Basis dieser Klasse nutzen:

```
Class Test1
    '...
End Class

Class Test2
    Private Sub Test()
        Dim test1 As Test1
    End Sub
End Class
```

Beispiele mit LINQPad 5

Für die folgenden, einführenden Beispiele in Visual Basic 2015 nutzen wir **LINQPad 5**, wo wir deutlich einfacher experimentieren können als in Visual

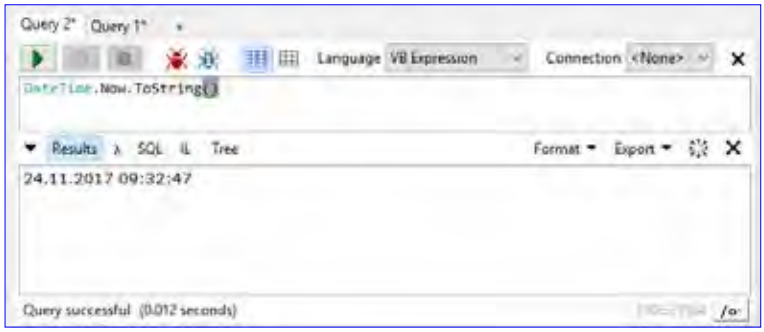


Bild 2: VB-Ausdrucks, dessen Ergebnis dann im Ausgabebereich landet

Bild 3: Eingabe einiger Anweisungen, die nacheinander ausgeführt werden.

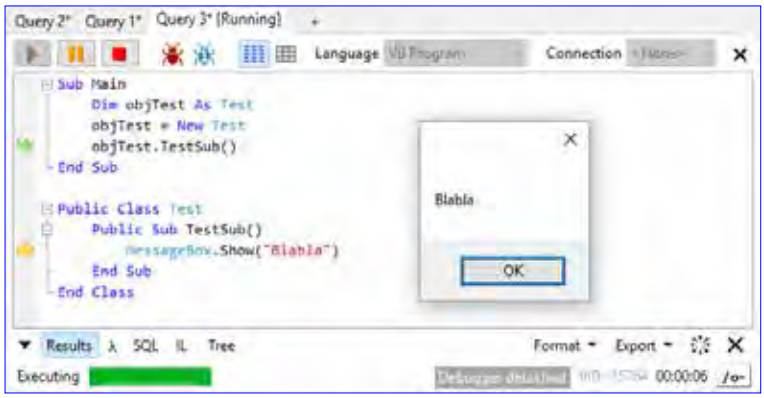


Bild 4: Ausführen der Methode einer Klasse

Studio, wo wir zumindest eine Konsolenanwendung brauchen, die wir immer wieder ausführen müssen.

Den Download von **LINQPad 5** finden Sie unter der Internetadresse **www.linqpad.net**.

Nach der Installation und dem Start von LINQPad 5 finden Sie im Hauptbereich ein Kombinationsfeld namens **Language**, mit der Sie die drei folgenden Optionen für Visual Basic einstellen können (siehe Bild 1):

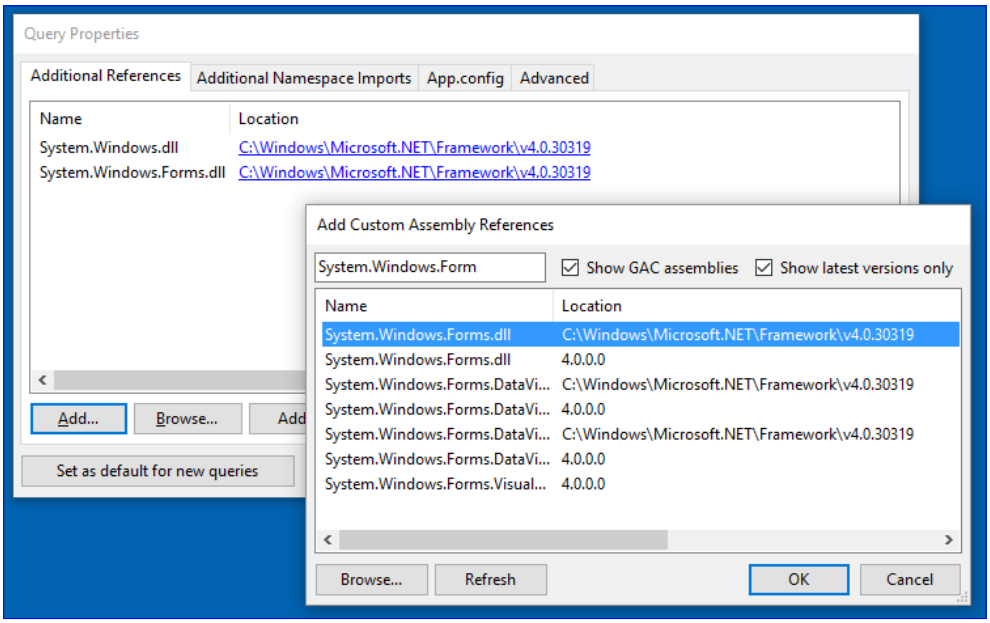


Bild 5: Hinzufügen von Bibliotheken

- **VB Expression**: Testen einfacher Ausdrücke wie etwa **DateTime.Now.ToString()** durch Eingabe und anschließende Betätigung von **F5** (siehe Bild 2)

- **VB Statement(s)**: Führt mehrere Anweisungen aus, also etwa den Inhalt einer Methode – nur eben ohne die übliche **Public Sub .../End Sub**-Zeilen (siehe Bild 3). Dazu ist wiederum das Betätigen der Taste **F5** beziehungsweise der Starten-Schaltfläche nötig.

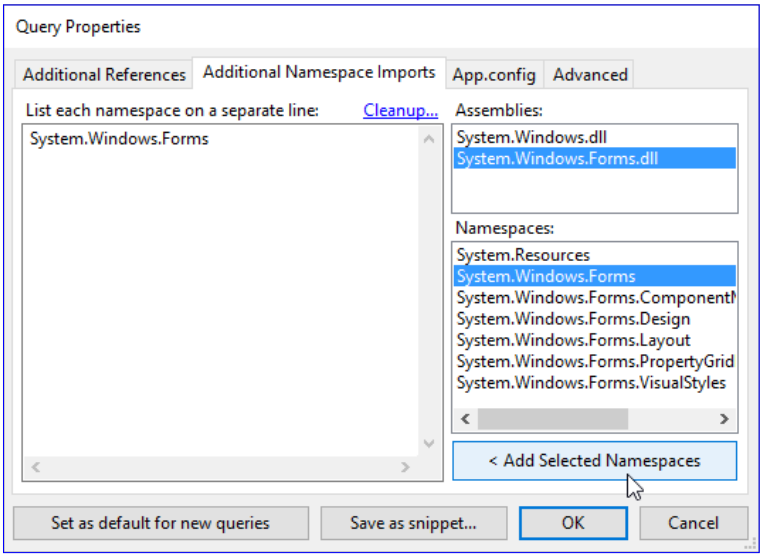


Bild 6: Hinzufügen von Namespaces

- **VB Program**: Hier können Sie komplette Methoden, Klassen und so weiter einbinden.
  Eine neue Registerseite dieses Typs blendet die Methode **Sub Main** ein, die beim Ausführen mit **F5** gestartet wird. Darunter können Sie beliebige weitere Elemente einbinden (siehe Bild 4). Mehr dazu in den folgenden Beispielen!

Namespaces hinzufügen

Unter C# fügen Sie zusätzliche Namespaces mit der **using**-Anweisung hinzu. Visual Basic nutzt stattdessen die **Imports**-Anweisung:

```
Imports System.Windows
```

In **LINQPad 5** können Sie diese Zeile in der aktuellen Version noch nicht nutzen. Dazu verwenden Sie einen kleinen Umweg:

- Klicken Sie mit der rechten Maustaste auf den Tab-Reiter und wählen Sie den Kontextmenü-Eintrag **Query Properties...** aus.
- Klicken Sie auf der Registerseite **Additional References** auf **Add...** und wählen Sie die hinzuzufügende Bibliothek aus, zum Beispiel **System.Windows.Forms.dll** (siehe Bild 5).
- Um dann den Namespace für den Code in **LINQPad 5** nutzen zu können, wechseln Sie auf die Seite **Additional Namespace Imports**. Klicken Sie hier rechts oben auf den Link **Pick from assemblies**. Dadurch wird ein weiterer Bereich eingeblendet, aus dem Sie die zu nutzenden Namespaces auswählen und mit der Schaltfläche **Add Selected Namespace** zur aktuellen Seite hinzufügen können (siehe Bild 6). Damit können Sie nun beispielsweise die **MessageBox**-Klasse nutzen.

Namespaces im Code definieren

Um einen Namespace im Code zu definieren, verwenden Sie genau dieselbe Anweisung wie unter C#:

```
Namespace Beispielnamespace
```

Unter LINQPad 5 können Sie diese Anweisung allerdings nicht nutzen – aber wenn Sie soweit sind, können Sie dies ja auch gleich im entsprechenden Projekt-Kontext testen.

Sub und Function

Genau wie unter VBA und im Gegensatz zu C# nutzen wir in Visual Basic 2015 die beiden Schlüsselwörter **Sub** und **Function**, um Prozeduren (und Methoden) sowie Funktionen zu definieren – das Schlüsselwort **void** von C# fällt hier also komplett weg. Beiden können Sie kein, ein oder mehrere Parameter übergeben. Die Funktion erhält am Ende wie unter VBA ein **As**-Schlüsselwort mit der Angabe des Datentyps für den Rückgabewert. Dieser gibt ihren Funktionswert zurück:

```
Public Function TestFunction() As String
    TestFunction = "bla"
End Function
```

Die Parameter werden wie unter VBA in einer kommaseparierten Liste in Klammern hinter dem Methodennamen übergeben, wobei dahinter das Schlüsselwort **As** und der Datentyp folgen:

```
Public Function TestFunction(strTest As String) As String
```

Datentypen

Unter Visual Basic 2015 gibt es, wie unter VBA, Werttypen, aber auch Verweistypen. Auch die einfachen Datentypen werden unter .NET bereits wie Objekte behandelt – zumindest, da diese eigene Eigenschaften bereitstellen. Hier sind die wichtigsten Datentypen für Visual Basic 2015 (in Klammern der Datentyp im .NET-Framework):

- **SByte**: -128 bis 127 (**System.SByte**), 8-Bit-Ganzzahl

Bubbling und Tunneling: Routed Events

Im Vergleich zu VBA, wo jedes Ereignis für das Steuerelement behandelt wurde, welches es auch ausgelöst hat, gibt es unter WPF einige Erweiterungen. Es gibt dort auch solche Ereignisse, aber in vielen Fällen werden Ereignisse an übergeordnete Elemente weitergeleitet. Das hört sich erstmal so an, als ob man es nicht unbedingt benötigt. Dennoch wollen wir das Prinzip anhand eines Beispiels erläutern, damit Sie mitunter auftretendes unerwartetes Verhalten von Code interpretieren können.

Unter WPF sind Elemente verschachtelt, das heißt, es gibt übergeordnete und untergeordnete Elemente. Das **Window**-Element ist normalerweise das Hauptelement, welches dann ein **Grid** enthält, in dem sich wiederum ein **Stackpanel** mit einer Schaltfläche befinden könnte:

```
<Window x:Class="BubblingAndTunneling.MainWindow" ... Title="MainWindow" Height="350" Width="525">
    <Grid>
        <StackPanel>
            <Button x:Name="btn" Content="Klick mich!" ... Click="btn_Click"></Button>
        </StackPanel>
    </Grid>
</Window>
```

Direkte Ereignisse/Direct Events

Wenn der Benutzer nun auf die Schaltfläche namens **btn** klickt, wird das Ereignis **Click** ausgelöst, welches üblicherweise durch eine Ereignismethode etwa namens **btn_Click** implementiert wird:

```
private void btn_Click(object sender, RoutedEventArgs e) {
    MessageBox.Show("Klick!");
}
```

Bei dem **Click**-Ereignis handelt es sich um ein direktes Ereignis. Dieses wird nur von dem Element ausgelöst, für das es implementiert wurde.

Tunneling Events

Bei Tunneling Events handelt es sich um alle Ereignisse, die mit dem Präfix **Preview** beginnen, also beispielsweise **PreviewMouseDown**. Was ist die Besonderheit dieser Ereignisse? Sie werden immer vom Element der obersten Ebene aus zu den unteren Ebenen abgearbeitet. Wenn Sie also das Ereignis **PreviewMouseDown** für alle vier Elemente des obigen Beispiels definieren, sieht das wie folgt aus:

```
<Window x:Class="BubblingAndTunneling.MainWindow" ... Title="MainWindow" Height="350" Width="525" MouseDown="Window_
MouseDown" PreviewMouseDown="Window_PreviewMouseDown">
```

```
    <Grid MouseDown="Grid_MouseDown" PreviewMouseDown="Grid_PreviewMouseDown" >
        <StackPanel MouseDown="StackPanel_MouseDown" PreviewMouseDown="StackPanel_PreviewMouseDown">
            <Button x:Name="btn" Content="Klick mich!" ... MouseDown="btn_MouseDown"
                PreviewMouseDown="btn_PreviewMouseDown"></Button>
        </StackPanel>
    </Grid>
</Window>
```

Für diese Ereignisse hinterlegen wir die folgenden Methoden (für **Debug.WriteLine** müssen Sie den passenden Namespace mit **using System.Diagnostics** hinzufügen):

```
private void Window_PreviewMouseDown(object sender, MouseButtonEventArgs e) {
    Debug.WriteLine("Window_PreviewMouseDown");
}
private void Window_MouseDown(object sender, MouseButtonEventArgs e) {
    Debug.WriteLine("Window_MouseDown");
}
private void Grid_PreviewMouseDown(object sender, MouseButtonEventArgs e) {
    Debug.WriteLine("Grid_PreviewMouseDown");
}
private void Grid_MouseDown(object sender, MouseButtonEventArgs e) {
    Debug.WriteLine("Grid_MouseDown");
}
private void StackPanel_PreviewMouseDown(object sender, MouseButtonEventArgs e) {
    Debug.WriteLine("StackPanel_PreviewMouseDown");
}
private void StackPanel_MouseDown(object sender, MouseButtonEventArgs e) {
    Debug.WriteLine("StackPanel_MouseDown");
}
private void btn_PreviewMouseDown(object sender, MouseButtonEventArgs e) {
    Debug.WriteLine("btn_PreviewMouseDown");
}
private void btn_MouseDown(object sender, MouseButtonEventArgs e) {
    Debug.WriteLine("btn_MouseDown");
}
```

Wenn wir nun die Anwendung starten und mit der linken Maustaste auf die Schaltfläche klicken, erhalten wir das Ergebnis aus Bild 1. Es werden also offensichtlich nur die Tunneling-Methoden ausgelöst, also diejenigen, die mit **Preview...** beginnen.

Immerhin ist hier gut zu erkennen, dass dies ausgehend von der obersten Ebene (**Window**) bis runter zum **Button**-Element geschieht. Aber warum werden die **MouseDown**-Ereignisse nicht ausgelöst?

Dies ist wiederum der Fall, wenn wir auf den freien Bereich neben der Schaltfläche klicken. Dann erhalten wir die folgende Ausgabe:

```
Window_PreviewMouseDown
Grid_PreviewMouseDown
StackPanel_PreviewMouseDown
StackPanel_MouseDown
Grid_MouseDown
Window_MouseDown
```

Hier werden also korrekt erst die Tunneling-Ereignisse ausgelöst und dann in umgekehrte Reihenfolge die Bubbling-Ereignisse.

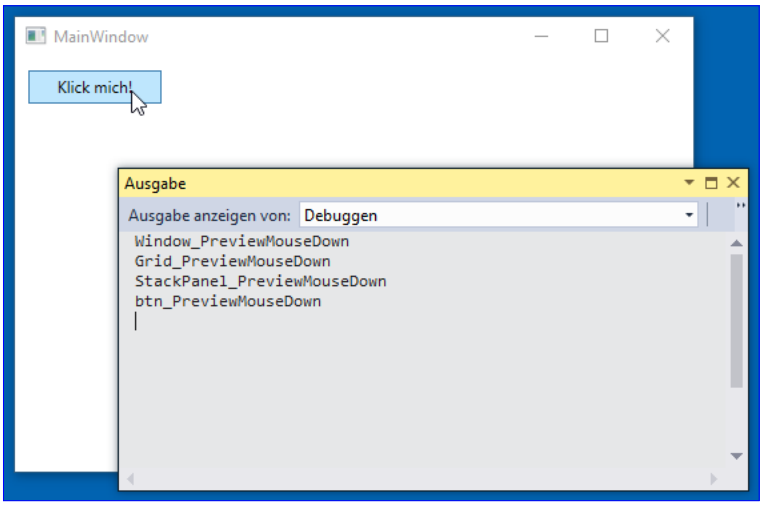


Bild 1: Es feuert nur die Hälfte der angegebenen Ereignisse.

Bubbling Events

Bevor wir uns darum kümmern, warum beim Anklicken der Schaltfläche nur die Tunneling-Events ausgelöst werden, noch kurz die Erläuterung der Bubbling-Ereignisse. Dies sind die Ereignisse, die in umgekehrter Reihenfolge wie die Tunneling-Ereignisse ausgelöst werden – also ausgehend vom auslösenden Element bis hin zum obersten Element der Hierarchie (wie gut im Beispiel oben zu sehen, wo wir nicht die Schaltfläche, sondern das **StackPanel**-Element angeklickt haben). Diese werden allerdings immer nach den Tunneling-Ereignissen abgearbeitet.

Warum kein MouseDown?

Nun schauen wir uns an, warum die Bubbling Events nicht ausgeführt werden, wenn wir das **Button**-Element mit der linken Maustaste anklicken. Der Grund ist einfach: Wenn Sie mit der linken Maustaste auf einen Button klicken, wird dessen **Click**-Ereignis ausgelöst, was dafür sorgt, dass das **MouseDown**-Ereignis nicht mehr feuert – und somit auch nicht die **MouseDown**-Ereignisse der in der Hierarchie über dem **Button**-Element liegenden Elemente. Dies gilt beispielsweise auch für das Textfeld.

Damit haben wir auch eine Erklärung, warum im Artikel **Drag and Drop** im Beispiel **Drag and Drop mit Dateien** die Bubbling Events nicht ausgeführt wurden und wir mit den Tunneling Events arbeiten mussten.

Elemente der Klasse RoutedEventArgs

Dem Kopf der Ereignismethoden können Sie als zweiten Parameter die Klasse **RoutedEventArgs** ausmachen. Diese liefert die folgenden Eigenschaften:

- **Handled**: Wenn Sie diesen Parameter auf den Wert **True** einstellen, wird das Routing der Ereignisse in der entsprechenden Ereignismethode unterbrochen. Ein Beispiel dazu finden Sie weiter unten.
- **RoutedEvent**: Liefert die **RoutedEvent**-Instanz.
- **Source**: Element, auf dem die Aktion ausgeführt wurde, die zum Routing führte.

Drag and Drop mit ListBox-Elementen

Im Artikel Drag and Drop-Grundlagen haben wir uns die grundlegenden Techniken für die Implementierung von Drag and Drop-Funktionen in WPF-Benutzeroberflächen angesehen. Nun gehen wir einen Schritt weiter und wollen Drag and Drop für das Fenster namens Versendungen unserer Beispielanwendung Bestellverwaltung umsetzen. Hier geht es dann nicht nur um einfaches Bewegen von Elementen per Maus, sondern auch um die Anpassung der dahinter stehenden Daten beziehungsweise Tabellen.

Wozu braucht man aber Drag and Drop so dringend? In der Tat lassen sich die meisten Tätigkeiten, die Sie damit ausführen können, auch auf andere Weise realisieren. Allerdings ist Drag and Drop und somit das einfache Ziehen von Elementen der Benutzeroberfläche mit der Maus doch ein sehr intuitiver Weg, um Aktionen durchzuführen. Ein Beispiel sind die beiden Listenfelder aus dem Artikel **m:n-Beziehung mit Listenfeld** (siehe Bild 1). Im Artikel haben wir Schaltflächen bereitgestellt, mit denen ein oder alle Artikel von einem Listenfeld zum anderen übertragen werden können. Nun wollen wir uns weiter vortasten und Drag and Drop-Funktionalität zu diesem Fenster hinzufügen. Zuvor schauen wir uns das jedoch an einigen grundlegenden Beispielen an.

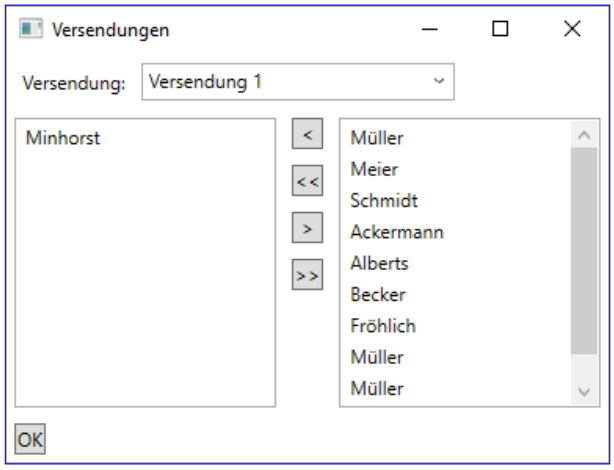


Bild 1: Für das Hin- und Herziehen von Einträgen zweier Listenfelder wäre Drag and Drop eine interessante Alternative.

Drag and Drop mit Listenfeldern

Nun schauen wir uns an, wie wir Drag and Drop in unser Beispiel mit den Versendungen einbauen. In der Beispielanwendung **Bestellverwaltung_SQLite** fügen wir dazu eine neue **Window**-Klasse namens **Versendungen_DragDrop** hinzu und kopieren den XAML-Code der Klasse **Versendungen** dort hinein. Passen Sie danach das **Window**-Element so an, dass die Klasse **Bestellverwaltung.Versendungen_DragDrop** referenziert wird und der Titel des Fensters angepasst ist:

```
<Window x:Class="Bestellverwaltung.Versendungen_DragDrop"
        xmlns="http://schemas.microsoft.com/winfx/2006/xaml/presentation"
        xmlns:x="http://schemas.microsoft.com/winfx/2006/xaml"
        xmlns:d="http://schemas.microsoft.com/expression/blend/2008"
        xmlns:mc="http://schemas.openxmlformats.org/markup-compatibility/2006"
        xmlns:local="clr-namespace:Bestellverwaltung"
        mc:Ignorable="d"
        Title="Versendungen Drag and Drop" Height="300" Width="400">
```

Dann kopieren Sie auch den Code der Code behind-Klasse von **Versendungen.xaml.cs** in **Versendungen_DragDrop.xaml.cs**. Ändern Sie hier zunächst in der folgenden Zeile Versendungen in **Versendungen_DragDrop**:

```
public partial class Versendungen_DragDrop : Window, INotifyPropertyChanged {
```

Der gleiche Schritt ist auch nochmal in der ersten Zeile der Konstruktor-Methode nötig:

```
public Versendungen_DragDrop() {
```

Danach können wir die Funktionen für das Drag and Drop hinzufügen.

Drag and Drop-Vorgang starten

Wir beginnen mit dem linken **ListBox**-Element. Hier legen wir gar nicht erst die Methode für das Ereignis **MouseDown** an, von dem wir dank der Experimente im Artikel **Drag and Drop-Grundlagen** schon wissen, dass es nicht feuert. Stattdessen legen wir direkt das entsprechende Tunneling-Ereignis **PreviewMouseDown** an:

```
<ListBox x:Name="lstAusgewaehlteKunden" Grid.Column="0" Grid.Row="1" Margin="5"
    ItemsSource="{Binding AusgewaehlteKunden}"
    DisplayMemberPath="Nachname"
    SelectedValuePath="ID" MouseDoubleClick="lstAusgewaehlteKunden_MouseDoubleClick"
        PreviewMouseDown="lstAusgewaehlteKunden_PreviewMouseDown"/>
```

Dafür hinterlegen wir nun die folgende Ereignismethode:

```
private void lstAusgewaehlteKunden_PreviewMouseDown(object sender, MouseButtonEventArgs e) {
    if (e.LeftButton == MouseButtonState.Pressed) {
        DragDrop.DoDragDrop(lstAusgewaehlteKunden, lstAusgewaehlteKunden.SelectedItem, DragDropEffects.Copy);
    }
}
```

Wenn wir die Anwendung nun starten, das Fenster **Versendungen_DragDrop** aufrufen und einen der Einträge der linken ListBox ziehen wollen, erhalten wir den Fehler aus Bild 2.

Da der Fehler nicht direkt an Ort und Stelle gemeldet wird, müssen wir eine Fehlerbehandlung hinzufügen:

```
private void lstAusgewaehlteKunden_PreviewMouseDown(object sender, MouseButtonEventArgs e) {
    try {
        if (e.LeftButton == MouseButtonState.Pressed) {
            DragDrop.DoDragDrop(lstAusgewaehlteKunden, lstAusgewaehlteKunden.SelectedItem, DragDropEffects.Copy);
        }
    }
    catch (System.Exception ex) {
        MessageBox.Show(ex.Message);
        throw;
    }
}
```

Dies liefert dann die Meldung aus Bild 3. Offensichtlich versuchen wir auf irgendeine Weise, ein Element des Typs **Null** zu ziehen. Welches das ist, erfahren wir per Debugging.

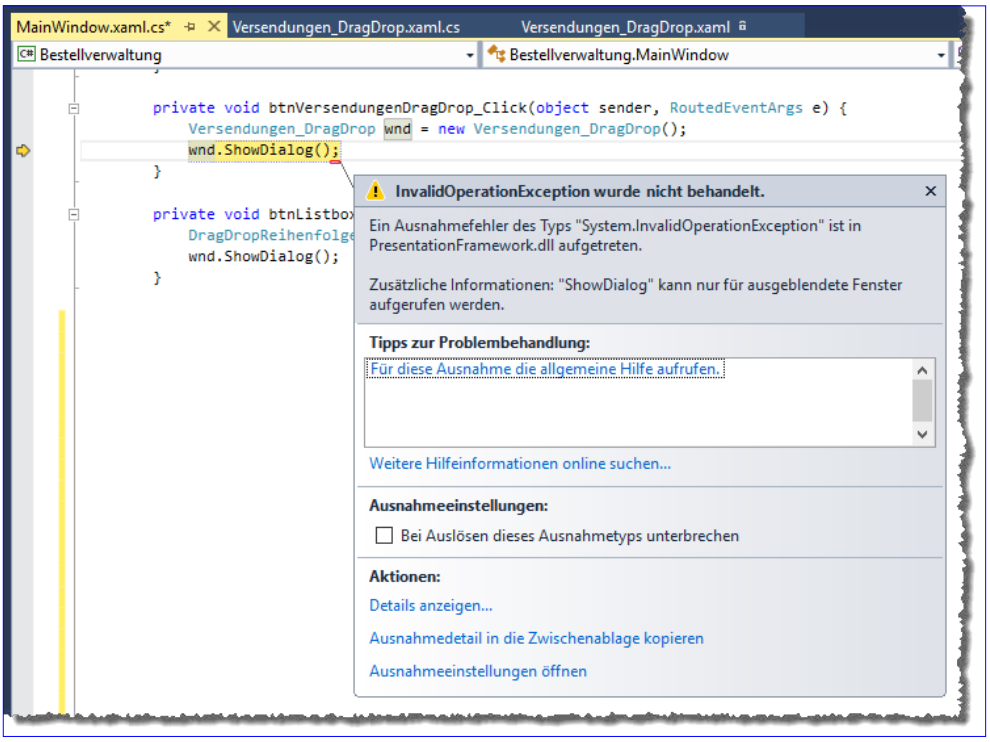


Bild 2: Fehler beim Versuch, ein Element aus der linken ListBox zu ziehen

Dazu setzen wir einen Haltepunkt in der ersten Zeile der Methode **lstAusgewaehlteKunden_PreviewMouseDown** und schauen uns den Inhalt der Parameter der Methode **DragDrop.DoDragDrop** an. Hier erfahren wir, dass die **SelectedItem**-Eigenschaft des **ListBox**-Elements den Wert **null** liefert (siehe Bild 4).

Wir müssen also offensichtlich einen ganz anderen Weg wählen, um beim Herunterdrücken der Maustaste Zugriff auf das dabei überfahrene Element zu bekommen. Dabei nutzen wir eine kleine Hilfsklasse, um das zu verschiebende Element zwischenzuspeichern:

```
public class DragDropData {
    public object ActualData { get; set; }
}
```

Dieses soll in der Eigenschaft **ActualData** das zu verschiebende Element speichern. Die Ereignismethode **lstAusgewaehlteKunden_PreviewMouseDown** ändern wir wie folgt ab:

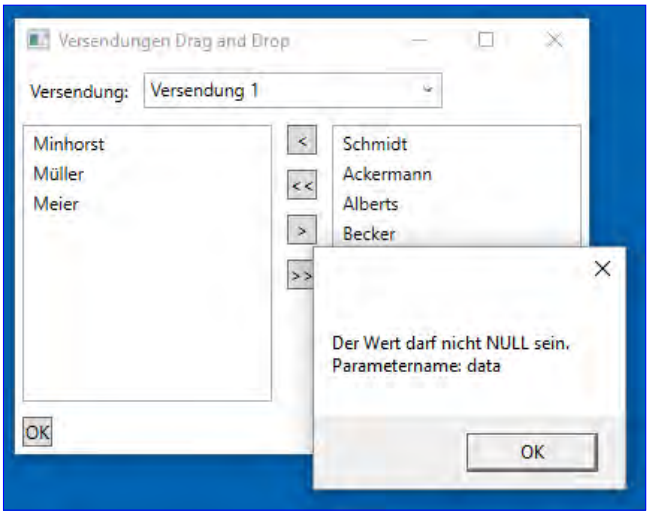


Bild 3: Der gleiche Fehler, diesmal mit Fehlerbehandlung

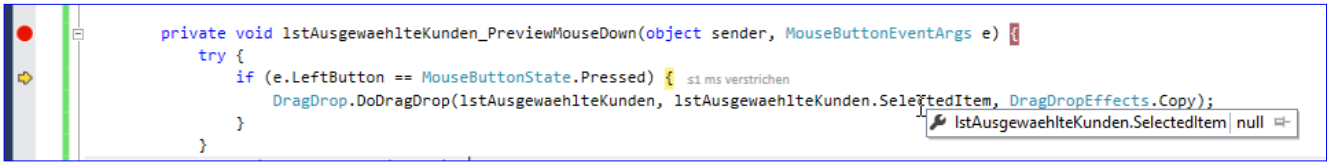


Bild 4: Der gleiche Fehler, diesmal mit Fehlerbehandlung

```
private void lstAusgewaehlteKunden_PreviewMouseDown(object sender, MouseButtonEventArgs e) {
    try {
        if (e.LeftButton == MouseButtonState.Pressed) {
            var item = (ListBoxItem)ItemsControl.ContainerFromElement(lst, (DependencyObject)e.OriginalSource);
            Kunde kunde = (Kunde)item.Content;
            DragDropData dragDropData = new DragDropData
            {
                ActualData = kunde,
            };
            DragDrop.DoDragDrop(lst, dragDropData, DragDropEffects.Move);
        }
    }
    catch (System.Exception ex) {
        MessageBox.Show(ex.Message);
        throw;
    }
}
```

Die Methode prüft zunächst, ob der Benutzer die linke Maustaste heruntergedrückt hat. In diesem Fall erstellt er ein neues Element des Typs **var** mit dem Namen **item**. Dies füllen wir mit dem **ListBoxItem**-Element, welches der Benutzer gerade in das andere Listenfeld ziehen möchte. Die Vorgehensweise, um an dieses Element zu kommen, ist nicht gerade intuitiv – zumindest nicht, wenn man noch nicht allzu tief im Objektmodell steckt: Wir ermitteln mit **e.OriginalSource** das Element, das der Benutzer mit der Maus ziehen möchte. In diesem Fall handelt es sich dabei etwa um ein **TextBlock**-Element, welches den Eintrag anzeigt. Dieses konvertieren wir in ein **DependendyObject**.

Dann ermitteln wir mit der **ContainerFromElement**-Methode der **ItemControl**-Klasse das Element der ListBox aus **lst**, welches das **TextBlock**-Element enthält – in diesem Fall das **ListBoxItem**-Element. Dieses konvertieren wir dann in ein **ListBoxItem**-Element. Der Inhalt des **ListBoxItem**-Elements sollte in unserem Fall ein Element des Typs **Kunde** sein, welches wir dann über die **Content**-Eigenschaft ermitteln, in **Kunde** konvertieren und in der Variablen **kunde** speichern. Dann erstellen wir eine neue Instanz unserer weiter oben vorgestellten Hilfsklasse **DragDropData** und weisen unser **Kunde**-Objekt **kunde** der einzigen Eigenschaft der Klasse zu. Damit können wir schließlich die **DoDragDrop**-Methode aufrufen und neben dem Verweis auf das **ListBox**-Element aus **lst** die Instanz der Klasse **DragDropData** übermitteln.

Wenn der Kunde nun das gezogene Element in der Ziel-ListBox abwirft, löst dies die **Drop**-Ereignismethode dieser ListBox aus. Diese sieht wie folgt aus:

```
private void lstAbgewaehlteKunden_Drop(object sender, DragEventArgs e) {
    DragDropData dragDropData = (DragDropData)e.Data.GetData(typeof(DragDropData));
    Kunde kunde = (Kunde)dragDropData.ActualData;
    KundeEntfernen(kunde);
}
```

Excel-Export mit Spreadsheet Light

Das Thema Reporting haben wir im DATENBANKENTWICKLER noch gar nicht behandelt. Das liegt auch daran, dass es unter C#/WPF keine so einfach einsetzbare Reporting-Funktion wie etwa die Berichte und Access gibt. Also wollen wir uns einmal verschiedene Möglichkeiten ansehen, um die Daten einer Datenbank aus einer C#-Anwendung zu exportieren, um diese etwa in einer anderen Anwendung zu öffnen und auszudrucken. Den Start machen wir mit Microsoft Excel.

Voraussetzungen

In diesem Artikel wollen wir uns nicht, wie etwas unter Access/VBA möglich, per Automation an Excel heranwagen und die Excel-Dokumente per Excel erstellen. Stattdessen nutzen wir eine Bibliothek namens **Spreadsheet Light**, welche kostenlos verfügbar ist und Excel-Dateien erstellen kann, ohne dass Excel überhaupt auf dem Rechner existiert. Das ist natürlich gerade für Webanwendungen interessant, da so keine Office-Installation benötigt wird, aber auch für Desktop-Anwendungen.

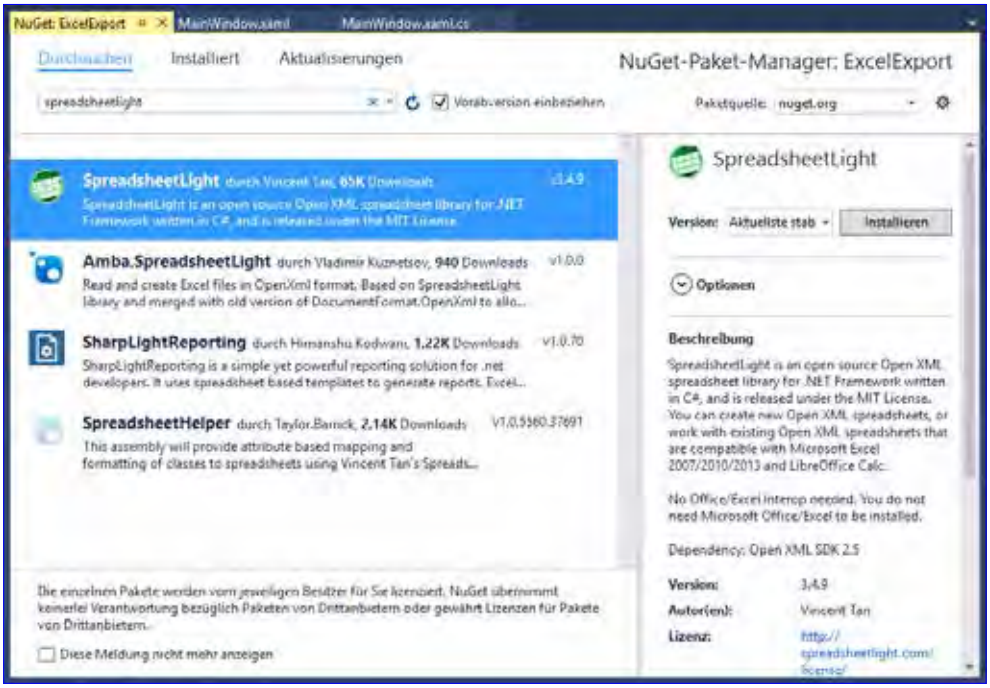


Bild 1: Hinzufügen von SpreadsheetLight über den Nuget-Paket-Manager

Spreadsheet light installieren

Spreadsheet light ist eine Open Source-Anwendung, die Sie kostenlos in Ihren Projekten dürfen. In den Beispielen verwenden wir ein C#/WPF-Projekt, dem Sie noch das entsprechende Paket plus Verweis hinzufügen müssen. Sie finden den Eintrag **SpreadsheetLight** im Nuget-Paket-Manager, den Sie über den Kontextmenü-Eintrag **Nuget-Pakete verwalten** des Projekts im Projektmappen-Explorer starten (siehe Bild 1).

Erste leere Excel-Datei erzeugen

Damit haben wir alle Voraussetzungen geschaffen. Ein erstes Beispiel wollen wir durch einen Mausklick auf eine neue Schaltfläche namens **btnDateiErstellen** ausführen. Diese fügen wir dem Fenster **MainWindow.xaml** hinzu und tragen für die Ereignisprozedur **btnDateiErstellen_Click** die folgenden beiden Befehle ein:

```
private void btnDateiErstellen_Click(object sender, RoutedEventArgs e) {
    SLDocument xls = new SLDocument();
    xls.SaveAs("LeereDatei.xlsx");
}
```

Der hier verwendete Typ **SLDocument** dürfte nun noch als fehlerhaft markiert werden. Das ist kein Problem – nachdem Sie den Verweis auf den Namespace **SpreadsheetLight** hinzugefügt haben, verschwindet die Markierung:

```
using SpreadsheetLight;
```

Wenn Sie das Projekt nun starten und die Schaltfläche betätigen, erstellt die Anwendung im Hintergrund ein erstes, leeres Excel-Dokument (siehe Bild 2). Das neue Dokument wurde übrigens im Verzeichnis der erstellte **.exe**-Datei erstellt, also beim Testen von Visual Studio aus etwa im Verzeichnis **bin\Debug** des Projektverzeichnisses.

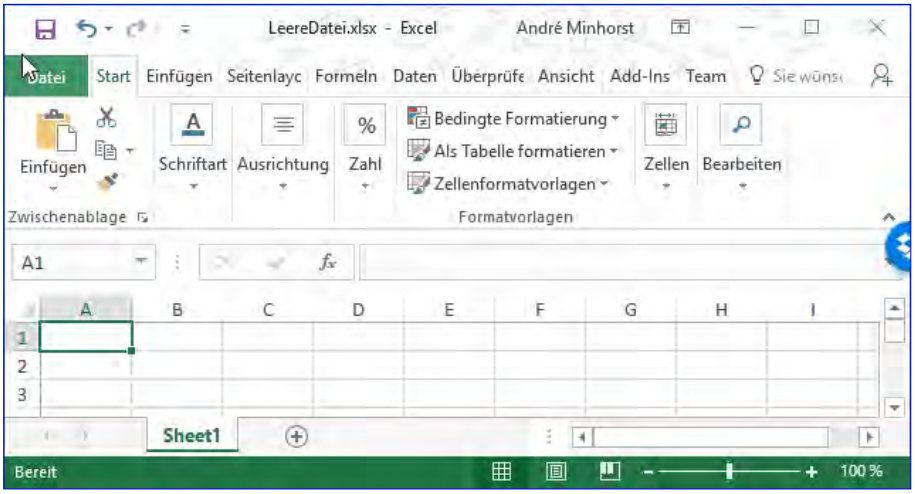


Bild 2: Die erste per SpreadsheetLight erstellt Excel-Datei

Wenn Sie Excel schon einmal von Access/VBA aus automatisiert haben, wissen Sie, dass Sie sonst einige Befehle mehr benötigen. Schauen wir uns nun an, wie wir die Excel-Datei füllen und formatieren können.

Daten zur Excel-Tabelle hinzufügen

Im zweiten Beispiel wollen wir eine Zelle der Tabelle mit einem Wert füllen, in diesem Fall einer Zeichenkette. Dazu verwenden wir die Methode **SetCellValue**, die als ersten Parameter die Angabe des Zellbezugs erwartet (hier **"A1"**) und als zweiten Parameter den einzufügenden Wert (**"BeispielText"**):

```
private void btnWertSchreiben_Click(object sender, RoutedEventArgs e) {
    SLDocument xls = new SLDocument();
    xls.SetCellValue("A1", "Beispieltext");
    xls.SaveAs("LeereDatei.xlsx");
}
```

Dabei markiert Visual Studio nun die Anweisung mit der Methode **SetCellValue** und liefert eine Fehlermeldung, nach der ein Verweis auf die Bibliothek **DocumentFormat.OpenXML** fehlt. Diese fügen Sie durch Auswahl des Menüeintrags **Projekt|Verweis hinzufügen...** und die Auswahl des Eintrags **DocumentFormat.OpenXML** im nun erscheinenden Dialog

Verweis-Manager hinzu (siehe Bild 3). Anschließend verschwindet auch diese Fehlermarkierung und Sie können das Einfügen des Zellinhalts nach dem Start des Debuggens und einem Klick auf die Schaltfläche **cmdWertSchreiben** ausprobieren.

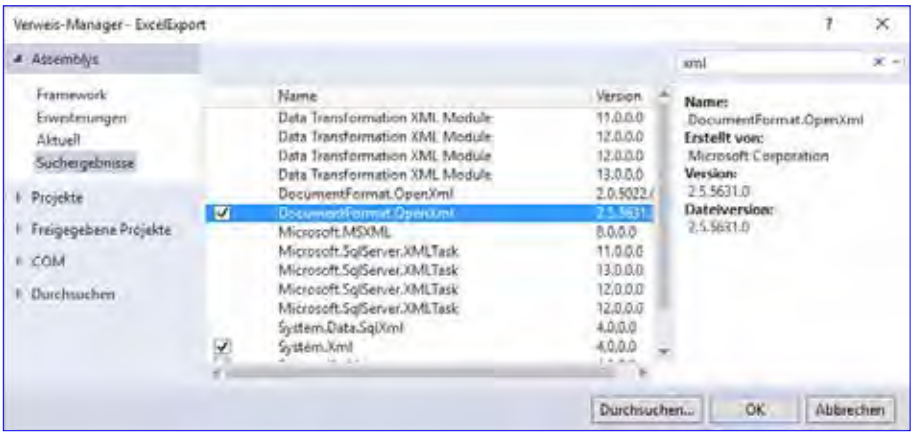


Bild 3: Verweis auf die Bibliothek **DocumentFormat.OpenXML**

Verschiedene Zellbezüge

Im ersten Beispiel haben wir als Zellbezug mit **A1** die klassische Variante gewählt. Excel zeigt ja auch die Zeilenüberschriften als Buchstaben und die Spaltenüberschriften als Zahlen an. Für die Programmierung ist dies indes nicht unbedingt immer hilfreich, vor allem, wenn Sie Daten innerhalb einer Schleife in die Excel-Tabelle schreiben wollen. Doch das ist unter **Spreadsheet Light** kein Problem: Die Methode **SetCellValue** bietet insgesamt 36 Überladungen. Warum so viele? Erstens, weil die Methode verschiedene Datentypen für den einzufügenden Wert entgegennehmen können soll, in diesem Fall **bool**, **byte**, **DateTime**, **decimal**, **InlineString**, **double**, **float**, **int**, **long**, **short**, **SLRstType**, **string, uint**, **ulong**, **ushort**, **DateTime** (mit **bool** für 1904-Epoche), **DateTime** (als **string**), **DateTime** (mit **bool** für 1904-Epoche als String). Das sind bereits 18 Varianten für den zweiten Parameter. Diese gibt es jeweils in Kombination mit dem **string**-Parameter **CellReference** (also beispielsweise **"A1"**) und mit den beiden **int**-Parametern **RowIndex** und **ColumnIndex** zur Angabe der zu füllenden Zelle als 1-basierten Index.

Wenn Sie also eine Matrix von 10x10 Zellen jeweils mit dem Produkt des Zellindexes füllen wollen, verwenden Sie die folgende Methode:

```
private void btnZehnMalZehn_Click(object sender, RoutedEventArgs e) {
    SLDocument xls = new SLDocument();
    for (int i = 1; i < 11; i++) {
        for (int j = 0; j < 11; j++) {
            xls.SetCellValue(i, j, i*j);
        }
    }
    xls.SaveAs("ZehnMalZehn.xlsx");
}
```

Diese Methode durchläuft zwei verschachtelte **for**-Schleifen mit dem Laufvariablen **i** und **j** und trägt jeweils das Produkt aus **i** und **j** in die Zelle mit dem entsprechenden Zeilen- und Spaltenindex ein (siehe Bild 4).

Zellen formatieren

Wollen wir nun ein Datum wie mit der folgenden Methode schreiben, erhalten wir nicht das erwartete Ergebnis:

```
private void btnDatumreihe_Click(object sender, RoutedEventArgs e) {
    SLDocument xls = new SLDocument();
    xls.SetCellValue(1, 1, new DateTime(2017,5,29));
    xls.SaveAs("DatumMitFormeln.xlsx");
}
```

Stattdessen erscheint in der Zelle der Wert **42884**, was der Anzahl der Tage seit dem 31.12.1899 entspricht. Im Gegensatz zu den bisher übergebenen Werten ist hier also eine Formatierung des Zellinhalts notwendig. Dies erledigen wir mit einer weiteren Methode namens **SetCellFormat**. Diese erwartet als ersten Parameter beziehungsweise für die ersten beiden Parameter die zu formatierende Zelle und als zweiten beziehungsweise dritten Parameter ein Objekt des Typs **SLStyle**. Dieses Objekt deklarieren und initialisieren wir zuerst und weisen seiner Eigenschaft **FormatCode** die Vorlage für das Format zu (in diesem Fall **"dd.mm.yyyy"**). Unsere Methode sieht nun wie folgt aus:

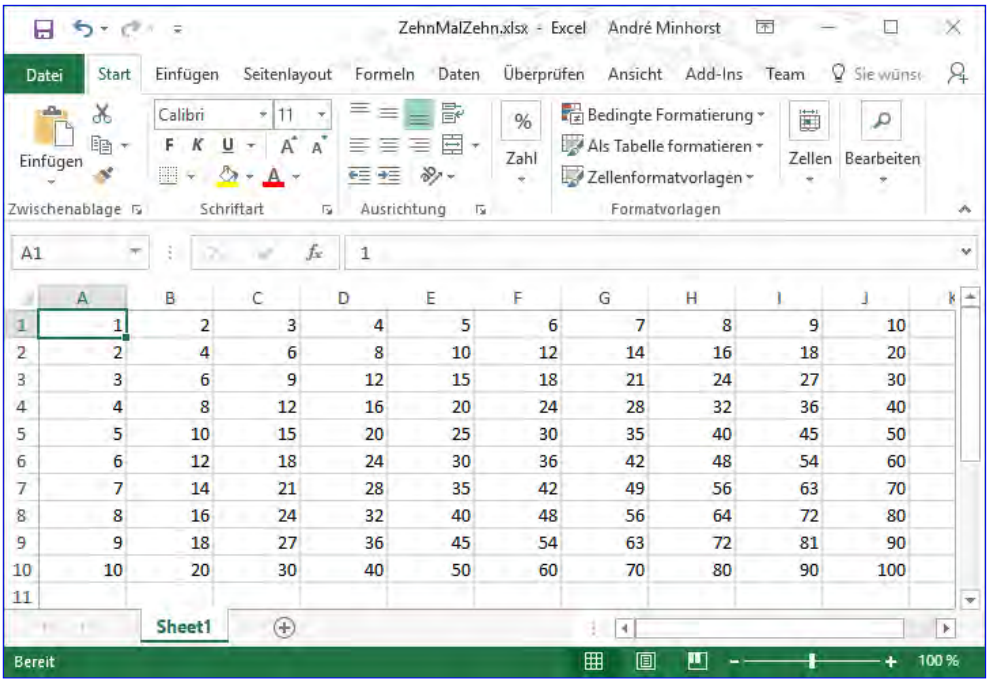


Bild 4: Excel-Datei, die per Zeilen- und Spaltenindex gefüllt wurde

```
private void btnDatumreihe_Click(object sender, RoutedEventArgs e) {
    SLDocument xls = new SLDocument();
    xls.SetCellValue(1, 1, new DateTime(2017,5,29));
    SLStyle style = new SLStyle();
    style.FormatCode = "dd.mm.yyyy";
    xls.SetCellStyle (1,1, style);
    xls.SaveAs("DatumMitFormeln.xlsx");
}
```

Spaltenbreiten und Zeilenhöhen

Damit erhalten wir nun auch eine entsprechend formatierte Zelle, die das Datum korrekt anzeigt. Allerdings ist die Spalte nun zu schmal, um das Datum wie gewünscht anzuzeigen. Die Spaltenbreite stellen Sie für jede Spalte individuell ein, und zwar mit der Methode **SetColumnWidth**:

Bestellverwaltung á la Visual Basic

In einer Umfrage haben uns viele Leser bescheinigt, dass Sie viel besser mit Visual Basic-Code arbeiten würden anstatt den großen Schritt von VBA auf C# zu wagen. Also wollen wir uns in diesem Artikel einmal ansehen, wie es aussieht, wenn wir unsere in den bisherigen Ausgaben entwickelte Beispielanwendung auf VB umstellen. Eines vorweg: Wir müssen nicht den kompletten Code anfassen, denn die Benutzeroberfläche haben wir ja vollständig mit XAML beschrieben. Aber auch der Rest liefert noch eine Menge Arbeit, wie die folgenden Seiten zeigen werden ...

Die erste Frage, die sich stellt, ist die nach der richtigen Vorgehensweise. Das C#-Projekt, das wir umwandeln möchten, besteht aus Code behind-Klassen mit C#-Code, die zu den **Window**- und **Page**-Objekten gehören, Klassendateien auf Basis von C# (**.cs**) und Konfigurationsdateien, die mit Sicherheit an der einen oder anderen Stelle festlegen, in welcher Sprache das Projekt programmiert wurde. So ist die **AssemblyInfo.cs**-Datei beispielsweise auch in der jeweiligen Sprache programmiert. Beginnen wir bei der **.sln**-Datei, welche die Lösung zusammensetzt – in unserem Fall ja nur aus dem reinen Anwendungsprojekt. Die dort referenzierte Projektdatei hat die Dateiendung **.csproj**, womit die C#-Affinität auch schon ausgeschöpft ist:

```
Project("{FAE04EC0-301F-11D3-BF4B-00C04F79EFBC}") = "Bestellverwaltung", "Bestellverwaltung\Bestellverwaltung.csproj",
"{FCD792BB-5FC8-4BC1-A66B-4165B450B126}"
```

In der **.csproj**-Datei finden sich beispielsweise die Referenzen, wozu auch eine auf die C#-Bibliothek gehört:

```
<Reference Include="Microsoft.CSharp" />
```

Danach folgen die Einträge für die einzelnen Klassen (**.xaml.cs**) und die dazugehörigen **.xaml**-Dateien sowie die übrigen **.cs**-Dateien, die zum Beispiel wie folgt aussehen:

```
<Compile Include="Kundenuebersicht.xaml.cs">
    <DependentUpon>Kundenuebersicht.xaml</DependentUpon>
</Compile>
```

Können wir nun also die entsprechenden Stellen in diesen Dateien ändern, die **.cs**-Dateien durch **.vb**-Dateien ersetzen und die Anwendung starten? So einfach ist es leider nicht. Wir finden beispielsweise keine Möglichkeit, etwa eine **.vb**-Klasse zum C#-Projekt hinzuzufügen. Selbst wenn wir eine neue VB-Klasse außerhalb des Projekts erstellen und es dann per Drag and Drop in den Projektmappen-Explorer zum Projekt hinzufügen, wird es zwar automatisch in der **.csproj**-Datei referenziert. Die darin enthaltene Klasse kann jedoch von den anderen C#-Klassen aus nicht referenziert werden.

Aus C# mach VB

Also legen wir ein neues VB-Projekt namens **Bestellverwaltung_VB** an. Dazu wählen Sie im Dialog **Neues Projekt** die Vorlage **Visual Basic\Windows\WPF-Anwendung** aus (siehe Bild 1).

Das VB-Projekt ist ähnlich aufgebaut wie ein C#-Projekt auf Basis der Vorlage WPF-Anwendung. Wir bekommen gleich die Klasse **MainWindow** zu Gesicht – allerdings hat die Code behind-Klasse diesmal mit der Dateiendung **.xaml.vb**.

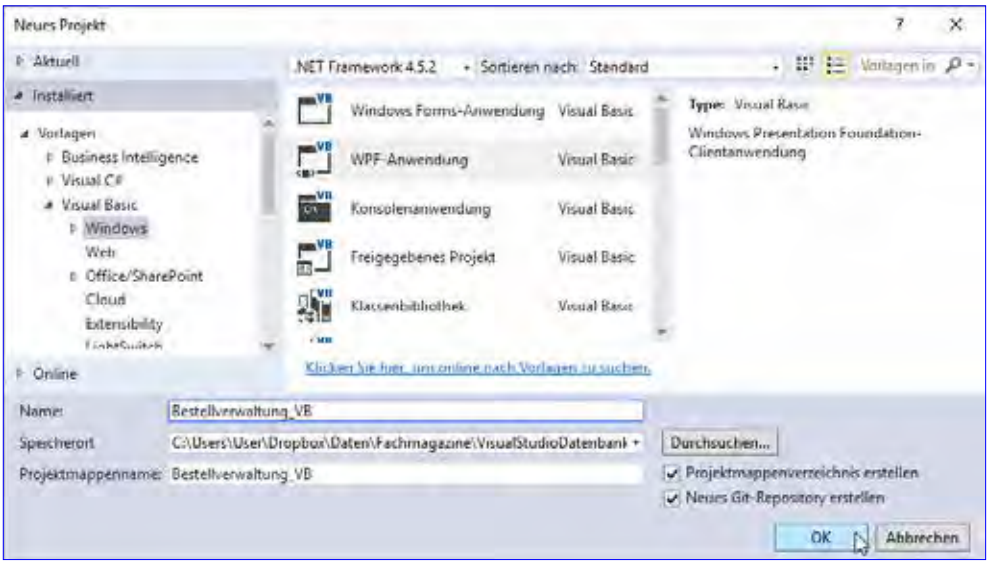


Bild 1: Anlegen des VB-Projekts

Konverter

Hier setzen wir direkt an und wollen die Code behind-Klasse des Fensters **MainWindow** der C#-Anwendung, also **MainWindow.xaml.cs**, in die Datei **MainWindow.xaml.vb** übertragen. Dazu verwenden wir einen automatischen Code-Konverter wie den auf der Webseite **converter.telerik.com**. Hier wählen Sie die gewünschte Übersetzung aus, hier also etwa C# to VB, und kopieren den zu übersetzenden Code in das linke Fenster. Nach einem Klick auf die Schaltfläche **Convert Code** wird dieser in das rechte Fenster konvertiert (siehe Bild 2).

Wie dies gelungen ist, wird sich zeigen, wenn wir nach und nach alle Elemente übersetzt haben. Dazu kopieren wir zunächst den Inhalt des rechten Fensters in die Klasse **MainWindow.xaml.vb**. Das liefert einige Fehlermeldungen, die zum größten Teil daraus resultieren, dass die dort referenzierten Klassen nicht vorhanden sind.

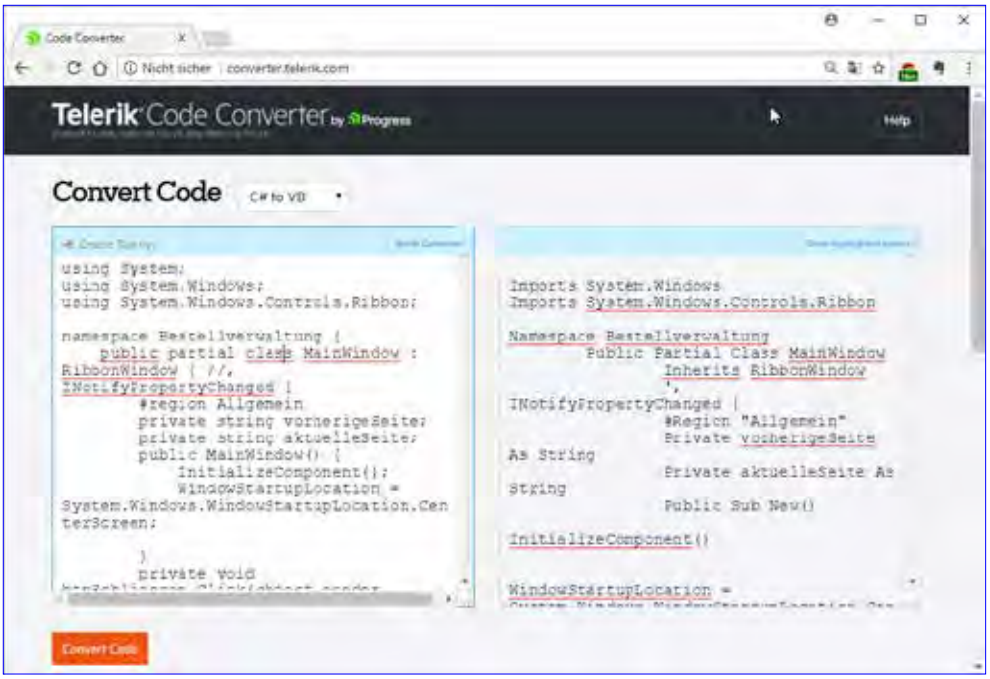


Bild 2: Automatisches Übersetzen des VB-Codes

Die erste können wir bereits beheben, indem wir einen Verweis auf die Klasse **System.Windows.Controls.Ribbon** hinzufügen, und zwar über den Dialog **Verweis-Manager**, den wir über den Menüeintrag **Projekt|Verweise** öffnen (siehe Bild 4).

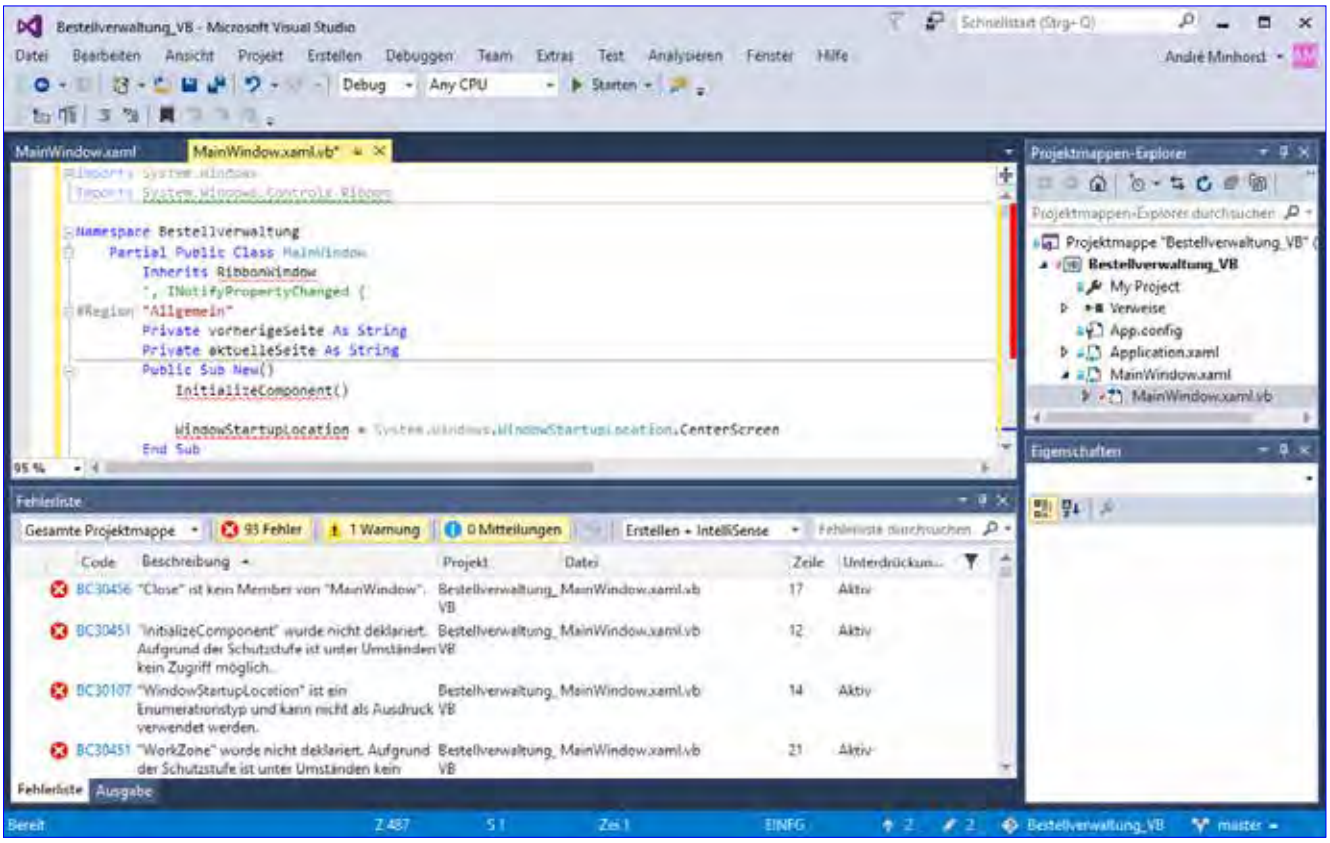


Bild 3: Beim Einfügen erscheinen einige Fehlermeldungen ...

Code temporär auskommentieren

Anschließend kommentieren wir fast den kompletten Code erst einmal aus. Dazu markieren Sie die auszukommentierenden Teile und betätigen die Tastenkombination **Strg + K, C**. Wir lassen nur die folgenden Zeilen übrig, um von dort aus Stück für Stück die entsprechenden Teile wieder in die Klasse einzubinden:

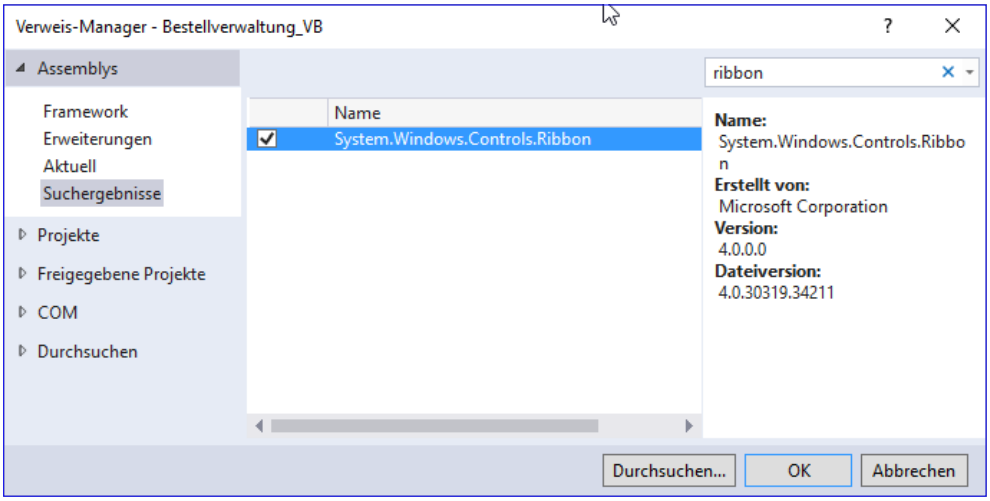


Bild 4: Verweis auf die Ribbon-Klasse hinzufügen

```
Imports System.Windows.Controls.Ribbon

Namespace Bestellverwaltung
```

```
    Partial Public Class MainWindow
        Inherits RibbonWindow
#Region "Allgemein"
        Private vorherigeSeite As String
        Private aktuelleSeite As String
        Public Sub New()
            InitializeComponent()
            WindowStartupLocation = System.Windows.WindowStartupLocation.CenterScreen
        End Sub
#End Region

    End Class
End Namespace
```

An dieser Stelle erhalten wir nur noch eine Fehlermeldung, und zwar für die Anweisung **InitializeComponent()**. Diese entfernen wir zunächst einfach.

XAML-Code kopieren

Der XAML-Code sollte keine Referenz auf die als Code behind-Datei verwendete **.cs**-Datei haben, sondern nur auf die Klasse beziehungsweise den Namespace. Also kopieren Sie einfach den kompletten Inhalt der Datei **MainWindow.xaml** des C#-Projekts in die gleichnamige Datei des neu erstellten VB-Projekts. Das Ergebnis sieht schon recht gut aus. Wir müssen allerdings noch die Bilddateien nachreichen, denn diese liegen ja noch nicht im Zielprojekt vor (siehe Bild 5). Um dies zu erledigen, würden wir nun am lieben einfach den Ordner **images** aus dem Projektmappen-Explorer des C#-Projekts in den des VB-Projekts ziehen. Das gelingt allerdings nicht.

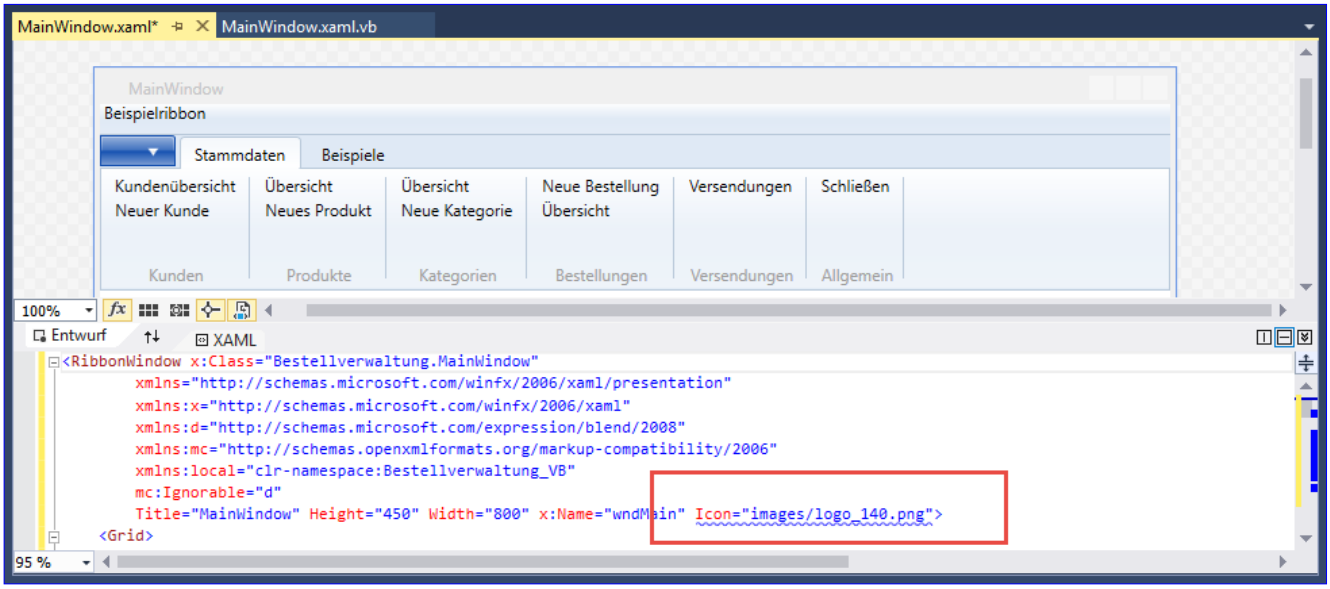


Bild 5: Die Bilddateien fehlen noch.

Bilder hinzufügen

Also ziehen wir einfach den Ordner **images** aus dem Windows Explorer in den Projektmappen-Explorer des VB-Projekts. Das gelingt, allerdings erscheinen die enthaltenen Bilddateien dort nicht! Allerdings wurde das Verzeichnis **images** im Projektordner im Dateisystem mit den enthaltenen Dateien angelegt. Deshalb ziehen wir die Bilder noch aus dem **images**-Verzeichnisses im Projektordner des VB-Projekts in den Ordner **images** im Projektmappen-Explorer. Danach erscheinen dort nicht nur die Bilder, sondern auch im Ribbon (siehe Bild 6).

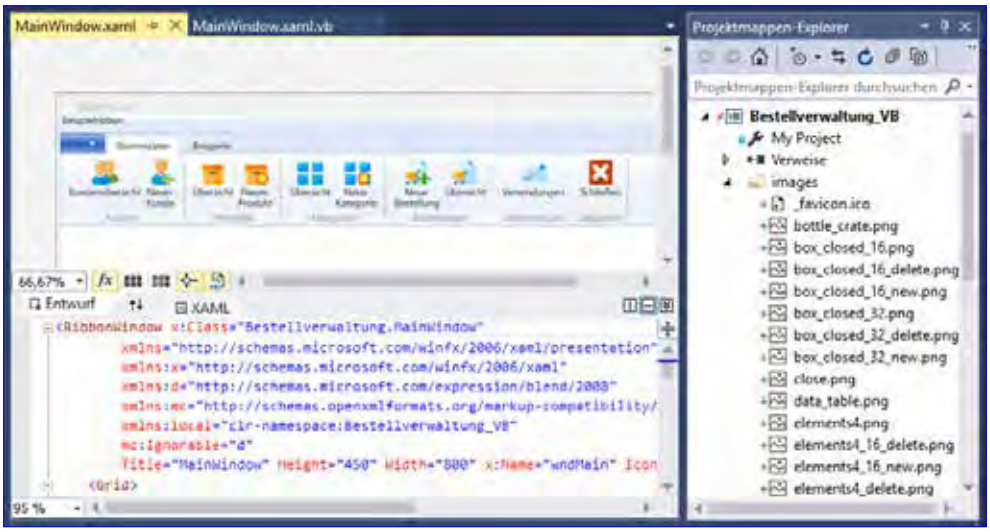


Bild 6: Anlegen der fehlenden Bilddateien

Versendungen-Fenster

Als Erstes nehmen wir nun das Fenster hinein, welches durch einen Klick auf die Ribbon-Schaltfläche **btnVersendungen** angezeigt wird. Dazu nehmen wir die Kommentarzeichen für die folgenden Zeilen in der Klasse **MainWindow.xaml.vb** wieder weg:

```
Private Sub btnVersendungen_Click(sender As Object, e As RoutedEventArgs)
    Dim wnd As New Versendungen()
    wnd.ShowDialog()
End Sub
```

Hier wird die Klasse **Versendungen** rot markiert, da diese ja noch gar nicht im Projekt enthalten ist. Also holen wird dies nach, indem wir eine neue **Window**-Klasse namens **Versendungen** anlegen. Dann kopieren wir den kompletten XAML-Code aus der Klasse **Versendungen.xaml** des C#-Projekts und fügen diesen in die neu erstellte Klasse im VB-Projekt ein. Sie brauchen hier nur den Namespace für das Attribut **xmlns:local** des **Window**-Elements auf **Bestellverwaltung_VB** zu ändern.

Dann kopieren wir auch hier einfach den Inhalt der Klasse **Versendungen.xaml.cs** in das linke Fenster der Webseite **http://converter.telerik.com/** und erzeugen mit einem Klick auf **Convert Code** den entsprechenden VB-Code. Daraus resultieren wiederum einige rote Markierungen und entsprechende Einträge in der Fehlerliste von Visual Studio (siehe Bild 7). Die meisten davon beziehen sich auf fehlende Objekte wie **Versendung**, **Kunde** oder **KundeVersendung**. Spätestens, wenn wir das nicht definierte Objekt **BestellverwaltungEntities** entdecken, wird dann klar, dass wir ja noch gar nicht das Entity Data Model zur Datenbank hinzugefügt haben.

Entity Data Model hinzufügen

Die zum Erstellen des Entity Data Model notwendigen Schritte in aller Kürze:

Drag and Drop-Grundlagen

Unter Access fehlen einige Features, die in anderen Programmiersprachen und Entwicklungsumgebungen zum guten Ton gehören. Eines davon ist die Drag and Drop-Funktionalität, die sich nur aufwendig abbilden ließ – und auch nur mit bestimmten ActiveX-Steuerelementen. Die eingebauten Steuerelemente wie Textfelder oder Listenfelder ließen leider kein natives Drag and Drop zu. Unter WPF und den .NET-Programmiersprachen sieht das ganz anders aus. Dieser Artikel liefert Grundlagen zu Drag and Drop.

Wozu braucht man aber Drag and Drop? In der Tat lassen sich die meisten Tätigkeiten, die Sie damit ausführen können, auch auf andere Weise realisieren. Allerdings ist Drag and Drop und somit das einfache Ziehen von Elementen der Benutzeroberfläche mit der Maus doch ein sehr intuitiver Weg, um Aktionen durchzuführen. Ein Beispiel sind die beiden Listenfelder aus dem Artikel **m:n-Beziehung mit Listenfeld** (siehe Bild 1). Im Artikel haben wir Schaltflächen bereitgestellt, mit denen ein oder alle Artikel von einem Listenfeld zum anderen übertragen werden können. Nun wollen wir uns weiter vortasten und Drag and Drop-Funktionalität zu diesem Fenster hinzufügen. Bevor wir uns das im Artikel **Drag And Drog mit ListBox**-Elementen ansehen, schauen wir uns einige grundlegende Beispielen an.

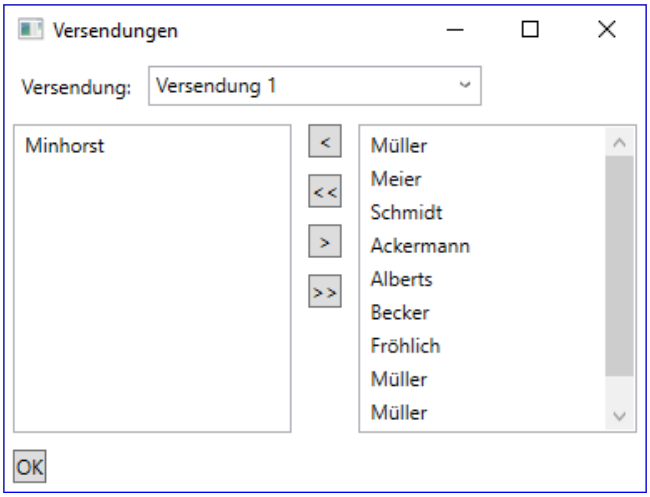


Bild 1: Für das Hin- und Herziehen von Einträgen zweier Listenfelder wäre Drag and Drop eine interessante Alternative.

Was ist Drag and Drop?

Drag and Drop nennen wir den Vorgang, bei dem der Benutzer ein Element der Benutzeroberfläche mit der linken Maustaste anklickt und diese dabei gedrückt hält. Dann wird die Maus mit dem Element zum Zielelement bewegt und das zu bewegende Element dort durch Loslassen der linken Maustaste an das Zielelement übergeben.

Dabei müssen Start- und Zielelement gar nicht unbedingt zwei verschiedene Elemente sein wie beim Beispiel mit den Listenfeldern. Es kann auch nur ein einziges Listenfeld sein, dessen Einträge Sie per Drag and Drop umsortieren wollen. Oder Sie bewegen die Elemente in einem **TreeView**-Steuerelement von einem übergeordneten Element zum nächsten.

Dazu benötigen wir ein paar Informationen. Zunächst müssen wir für das abgebende Element festlegen, dass es das Ziehen der enthaltenen Elemente mit der Maus überhaupt erlaubt. Dann wollen wir uns irgendwie merken, welches Element wir mit der Maus angepackt haben, um es zu verschieben (oder auch zu kopieren). Schließlich soll das aufnehmende Element mit der Möglichkeit zum Aufnehmen des zu übergebenden Elements ausgestattet werden, also als Drag and Drop-Ziel verfügbar gemacht werden. Da man unter Windows eine ganze Menge Dinge durch die Gegend ziehen kann – zum Beispiel von Windows Explorer auf verschiedene andere Ziele – müssen Sie dem Zielelement auch noch mitteilen, auf welche zu übergebenden Elemente es überhaupt reagieren soll, indem es das Symbol als mögliches Zielelement einblendet. Schließlich benötigen wir verschiedene

Ereignisse, für die wir passende Methoden implementieren, um das Verhalten von Drag and Drop-Quelle und -Ziel festzulegen.

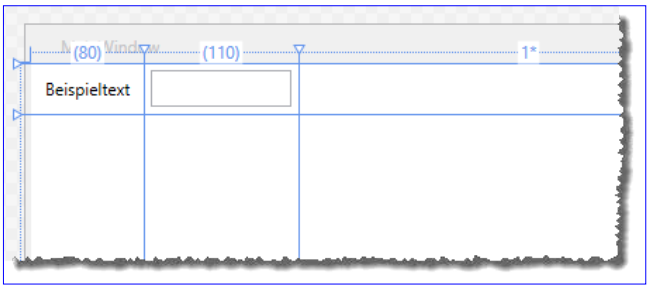


Bild 2: Label und TextBox als erstes Drag and Drop-Beispiel

Drag and Drop ist auch eine Alternative zum Kopieren/Ausschneiden und Einfügen. Der wesentliche Unterschied neben den dazu notwendigen Techniken (Ziehen mit der Maus bei Drag and Drop auf der einen, Tastenkombinationen wie **Strg + X**, **V** oder **C** und Kontextmenü-Einträge auf der anderen Seite) ist der Ort, an dem wir uns merken, welches Element von A nach B bewegt werden sollen. Beim Kopieren/Ausschneiden und Einfügen ist dies die Zwischenablage. Bei Drag and Drop benötigen wir eine andere Möglichkeit, in diesem Fall ein Element des Typs **DataObject**.

Einfaches Beispiel: Bezeichnungsfeld in Textfeld ziehen

Wir schauen uns zu Beginn ein einfaches Beispiel an, bei dem wir die Beschriftung eines Bezeichnungsfeldes einfach in ein Textfeld ziehen wollen. Dazu legen wir entsprechende Steuerelemente in einem neuen Fenster namens **MainWindow.xaml** eines neuen Projekts an (s. Bild 2). Die beiden Steuerelemente definieren wir wie wie folgt:

```
<TextBox x:Name="txtDrop" Grid.Column="1" Grid.Row="1" Margin="5" Drop="txtDrop_Drop" Width="100" />
<Label x:Name="lblDrag" Content="Beispieltext" Grid.Column="0" Grid.Row="1" Margin="5" MouseDown="lblDrag_MouseDown" />
```

Unter Visual Basic sieht die Ereignismethode für das Ereignis **MouseDown** des Bezeichnungsfeldes wie folgt aus:

```
'Visual Basic
Private Sub lblDrag_MouseDown(sender As Object, e As MouseButtonEventArgs)
    DragDrop.DoDragDrop(lblDrag, lblDrag.Content, DragDropEffects.Copy)
End Sub
```

Wir benötigen also nur eine einzige Zeile, um den Vorgang zu starten! Dabei rufen wir die Methode **DoDragDrop** der Klasse **DragDrop** auf. Diese erwartet drei Parameter:

- **dragSource**: Quelle des Drag and Drop-Vorgangs, also beispielsweise ein Steuerelement wie in diesem Fall das Bezeichnungsfeld
- **data**: Die per Drag and Drop bewegten Daten werden mit dem Parameter **data** in einen Zwischenspeicher ähnlich der Zwischenablage gegeben und können dort beim Drop wieder abgerufen werden.
- **allowedEffects**: Effekte sind verschiedene Darstellungsarten des Mauszeigers.

Im vorliegenden Fall übergeben wir für den ersten Parameter einen Verweis auf das Bezeichnungsfeld **lblLabel**. Der zweite Parameter nimmt den Inhalt der Eigenschaft **Content** des Bezeichnungsfeldes auf, hier eine einfache Beschriftung. Da es sich

hierbei um ein reines Kopieren von Daten handelt, nämlich das Kopieren der Beschriftung des Bezeichnungsfeldes in das Textfeld, soll nur das Symbol für das Kopieren mit der Maus angezeigt werden (siehe Bild 3).

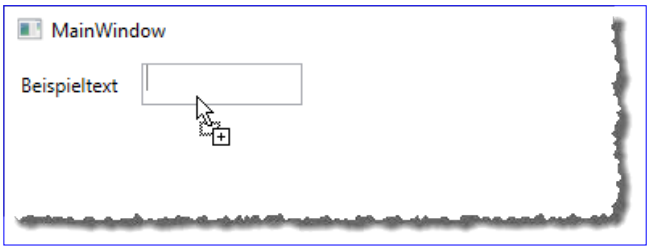


Bild 3: Veränderter Mauszeiger beim Drag and Drop

Damit kommen wir Fallenlassen des zu kopierenden Inhalts über dem Zielobjekt. Hier verwenden wir nun das Ereignis Drop des Textfeldes, das wir wie folgt implementieren:

```
'Visual Basic
Private Sub txtDrop_Drop(sender As Object, e As DragEventArgs)
    Dim str As String
    str = e.Data.GetData(DataFormats.StringFormat)
    txtDrop.Text = str
End Sub
```

Die Methode liefert die gewohnten beiden Parameter, wobei der zweite den Typ **DragEventArgs** hat (mehr dazu weiter unten). In diesem Fall nutzen wir das Objekt **data** der mit dem Parameter e übergebenen **DragEventArgs**-Klasse, dessen Methode **GetData** mit dem Parameter **DataFormats.StringFormat** den gespeicherten Text zurückgibt. Diesen speichern wir in der String-Variablen **str** und weisen diese dann der **Text**-Eigenschaft des Textfeldes zu. Das Ergebnis sieht schließlich wie in Bild 4 aus.

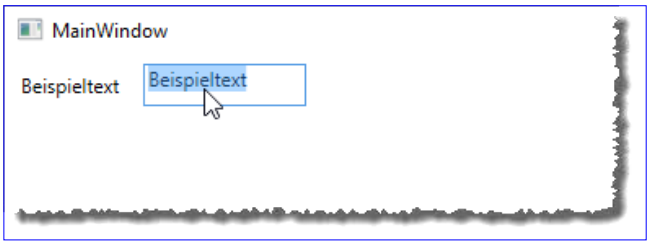


Bild 4: Erfolgreicher Drag and Drop-Vorgang

Zusammengefasst haben wir für den Start von Drag and Drop das Ereignis **MouseDown** genutzt, dort den Drag and Drop-Vorgang gestartet und dadurch auch das Ereignis **Drop** aktiviert. Ohne den Aufruf von **DoDragDrop** können Sie nämlich mit der Maus ziehen und fallenlassen, was sie möchten – das **Drop**-Ereignis wird nicht ausgelöst.

Verfeinerung: Maustaste prüfen

Im aktuellen Zustand können Sie auch mit der rechten Maustaste den Drag and Drop-Vorgang ausführen. Das liegt daran, dass wir in der Methode **lblDrag_MouseOver** nicht geprüft haben, welche Maustaste gerade gedrückt wurde. Dies holen wir nun nach und passen die Methode wie folgt an:

```
Private Sub lblDrag_MouseDown(sender As Object, e As MouseButtonEventArgs)
    If e.LeftButton = MouseButtonState.Pressed Then
        DragDrop.DoDragDrop(lblDrag, lblDrag.Content, DragDropEffects.Copy)
    End If
End Sub
```

Hier prüfen wir vorab, ob die linke Maustaste gedrückt ist. Damit wird das Drag and Drop mit gedrückter rechter Maustaste unterbunden.

Drag and Drop aus anderen Anwendungen

Wenn Sie einen Text aus einer anderen Anwendung in ein Textfeld einer WPF-Anwendung ziehen wollen, klappt das mit der gegebenen Konfiguration ebenfalls. Sie können nun beispielsweise Text aus einem Text-Editor oder aus Word markieren und diesen dann in das Textfeld ziehen. Aber: Dazu wären die bisherigen Schritte gar nicht nötig gewesen – Sie können auch einfach so den Text aus einer externen Anwendung in das Textfeld ziehen. Dazu müssen Sie noch nicht einmal ein Attribut des Textfeldes anpassen. Interessant ist noch die Frage: Feuert denn das Drop-Ereignis des Textfeldes, wenn Text aus einer externen Anwendung abgelegt wird? Die Antwort lautet nein.

Drag and Drop in andere Anwendungen

Bei Texten gelingt das Drag and Drop aus einem Textfeld in eine andere Anwendung, welche als Drop-Ziel ausgelegt ist wie etwa Microsoft Word, ohne weiteres Zutun.

Auch wenn man einen Drag and Drop-Vorgang in der WPF-Anwendung so beginnt, dass der zu verschiebende Inhalt in die Drag and Drop-Zwischenablage übertragen wird, können Sie den Inhalt etwa in eine Textverarbeitung übertragen. Auf diese Weise können Sie etwa den Text der Schaltfläche aus dem ersten Beispiel in eine Textverarbeitung statt in das Textfeld ziehen.

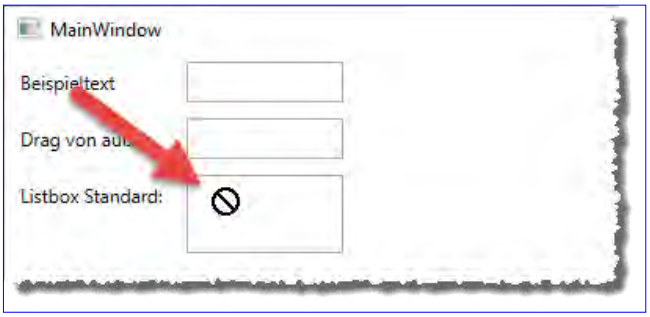


Bild 5: Drag and Drop auf eine Standard-ListBox

Drag and Drop in eine ListBox

Textfelder sind also immer für Drag and Drop empfänglich, wenn es sich um Daten im passenden Typ handelt, also um Texte. Was aber, wenn wir einen Text auf ein anderes Steuerelement droppen – beispielsweise ein **ListBox**-Steuerelement? Dazu fügen wir unserem Beispielformular eine ListBox hinzu und nennen diese **lstDrop**:

```
<ListBox x:Name="lstDropStandard" Grid.Column="1" Grid.Row="3" Margin="5" Width="100" Height="50" />
```

Wenn wir den Text des Bezeichnungsfeldes **lblDrag** auf das **ListBox**-Element ziehen, wird der Mauszeiger in ein Symbol umgewandelt, dass deutlich macht, dass das **ListBox**-Element kein valides Ziel für die Drag and Drop-Anweisung ist (siehe Bild 5). Also stellen wir nun das Attribut **AllowDrop** auf **True** ein:

```
<ListBox x:Name="lstDropStandard" ... AllowDrop="True" />
```

Damit erscheint schon einmal das richtige Symbol, wenn Sie das **ListBox**-Element beim Drag and Drop mit der Maus überfahren. Beim Fallenlassen geschieht jedoch nichts. Kein Wunder: Ein **ListBox**-Element nimmt seine Inhalte ja auch über ganz andere Wege entgegen – zum Beispiel über die **Add**-Methode der **Items**-Auflistung. Diese implementieren wir nun für die **Drop**-Methode des **ListBox**-Element, das wir dazu als neues Element wie folgt erweitern:

```
<ListBox x:Name="lstDropListBox" AllowDrop="True" Drop="lstDropListBox_Drop"></ListBox>
```

Die Drop-Methode implementieren wie folgt:

```
Private Sub lstDropListBox_Drop(sender As Object, e As DragEventArgs)
    Dim str As String
    str = e.Data.GetData(DataFormats.StringFormat)
    lstDropListBox.Items.Add(str)
End Sub
```

Das Ergebnis sieht wie in Bild 6 aus. Hier haben wir nun die Verarbeitung von Drag and Drop implementiert, ohne dass wir zuvor die Methode **DoDragDrop** aufgerufen haben. Dafür haben wir allerdings das Attribut **AllowDrop** aktiviert und das Ereignis **Drop** implementiert. Die dafür angelegte Methode liest den Inhalt des Data-Objekts und fügt diesen als neues Element an das **ListBox**-Element an.

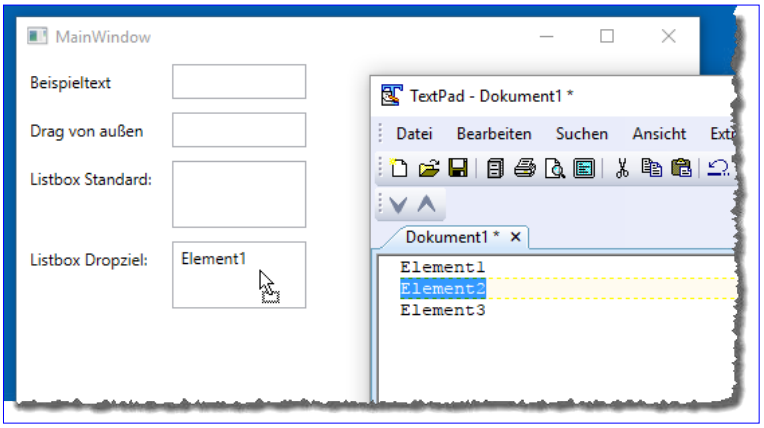
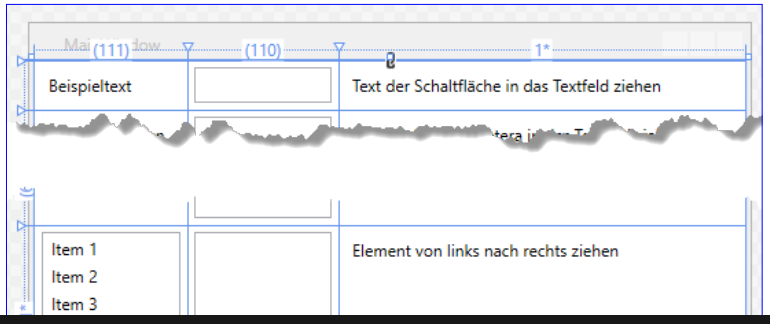


Bild 6: Drag and Drop in eine ListBox

Festlegen, ob ein Element ein geeignetes Ziel für Drag and Drop ist

Nun haben wir in ein paar Beispielen Texte hin und her verschoben, aber wir haben uns noch nicht darum gekümmert, ob das jeweilige Zielelement überhaupt ein gültiges Ziel ist - und wie wir dem Benutzer anzeigen, dass es sich hierbei um ein Ziel handelt. Wann können wir nun feststellen,

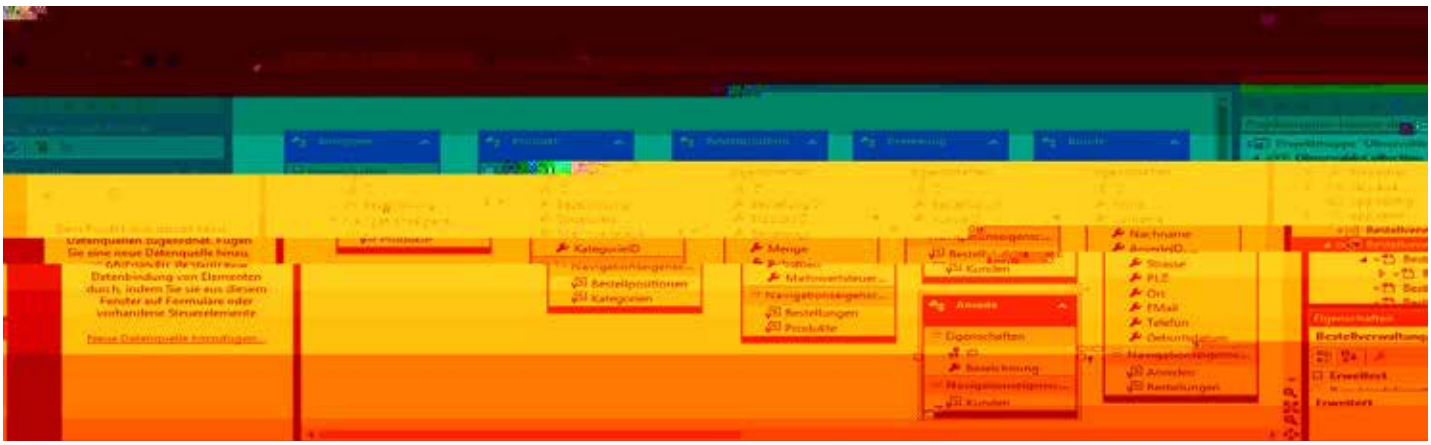


Die Leseprobe dieses Artikels ist hier zu Ende.
Willst Du mehr?
ZUM SHOP
2.400 Seiten Know-how rund um die Datenbankentwicklung mit .NET
KOMPENDIUM
DATENBANK ENTWICKLER
KOMPENDIUM
André Minhorst Verlag
Spare 20 EUR mit Code dbew20

AUSGABE 06/2017

DATENBANK ENTWICKLER

MAGAZIN FÜR DIE DATENBANKENTWICKLUNG MIT VISUAL STUDIO FÜR DESKTOP, WEB UND CO.



TOP-THEMEN:

VISUAL STUDIO	Vorlagen erstellen	SEITE 3
WPF-BASICS	Grid im Griff	SEITE 15
VB-BASICS	Bedingungen und Schleifen	SEITE 22
INTERAKTIV	E-Mails mit Outlook oder SMTP senden	SEITE 33
LÖSUNGEN	Brief mit Word erstellen	SEITE 56

Visual Studio-Vorlagen erstellen

Wer die bisherigen Artikel verfolgt und zuvor mit Access gearbeitet hat, stellt fest, dass es einige Aufgaben gibt, die sich unter Access einfacher durchführen ließen – zum Beispiel das Ausstatten eines Formulars mit einer Datenherkunft und das Hinzufügen der gebundenen Steuerelemente zu diesem Formular. Nun gibt es auch unter Visual Studio die Möglichkeit, Assistenten selbst zu programmieren. Damit könnten wir dann selbst die Helferlein erstellen, die uns bei Routineaufgaben unterstützen – wie beispielsweise das Anlegen eines Window- oder Page-Elements auf Basis einer Entität. Doch bis dahin ist es noch ein langer Weg – in diesem Artikel schauen wir uns zunächst an, wie Sie einfache Vorlagen für Visual Studio erstellen.

Standards erweitern

Microsoft-Produkte wie das Office-Paket inklusive Access oder auch Visual Studio sind tolle Produkte, die schon eine Menge Funktionen bieten und den Benutzer bestmöglich bei seinen Aufgaben unterstützen.

Bild 1: Standard-Vorlagen

Allerdings gelingt dies nicht immer so, wie man sich das vorstellt – was allerdings zum größten Teil daran liegt, dass es einfach so viele unterschiedliche Aufgaben gibt. Die Entwickler von Microsoft können einfach nicht alle denkbaren Anforderungen der Benutzer ihrer Programme vorausahnen – deshalb kommen die meisten Vorlagen in einer Minimalausstattung, die der Entwickler für seine Zwecke erweitern kann (siehe Bild 1). Das ist aber auch gar nicht weiter schlimm, denn sowohl für Office als auch für Visual Studio gibt es verschiedene Möglichkeiten, die Anwendungen um Vorlagen, benutzerdefinierte Add-Ins oder Wizards zu erweitern. Wir schauen uns in diesem Artikel an, wie Sie aus Ihren selbst gestalteten **Window**-, **Page**-, **Class**- und sonstigen Elementen Vorlagen erstellen können, die Sie dann beim Anlegen neuer Elemente nutzen können.

Beispiele für Vorlagen

Wenn Sie mit WPF arbeiten und Windows-Anwendungen entwickeln, haben Sie vielleicht bestimmte Einstellungen, die für alle Steuerelemente verwendet werden sollen, wenn diese nicht gerade im Steuerelement selbst überschrieben werden. Solche Eigenschaften legt man beispielsweise wie folgt in einem Element namens **Window.Resources** an:

```
<Window.Resources>
    <Style TargetType="Button">
        <Setter Property="Margin" Value="3"></Setter>
        <Setter Property="Padding" Value="5"></Setter>
        <Setter Property="VerticalAlignment" Value="Top"></Setter>
        <Setter Property="HorizontalAlignment" Value="Left"></Setter>
    </Style>
</Window.Resources>
```

Wenn Sie solche Einstellungen immer wieder verwenden, müssen Sie diese jeweils in neue **Window**- oder **Page**-Element integrieren. Sie können aber auch eine Vorlage erstellen, der Sie die gewünschten Elemente hinzufügen, und diese dann bei Bedarf statt des herkömmlichen **Window**-Elements zum Projekt hinzufügen.

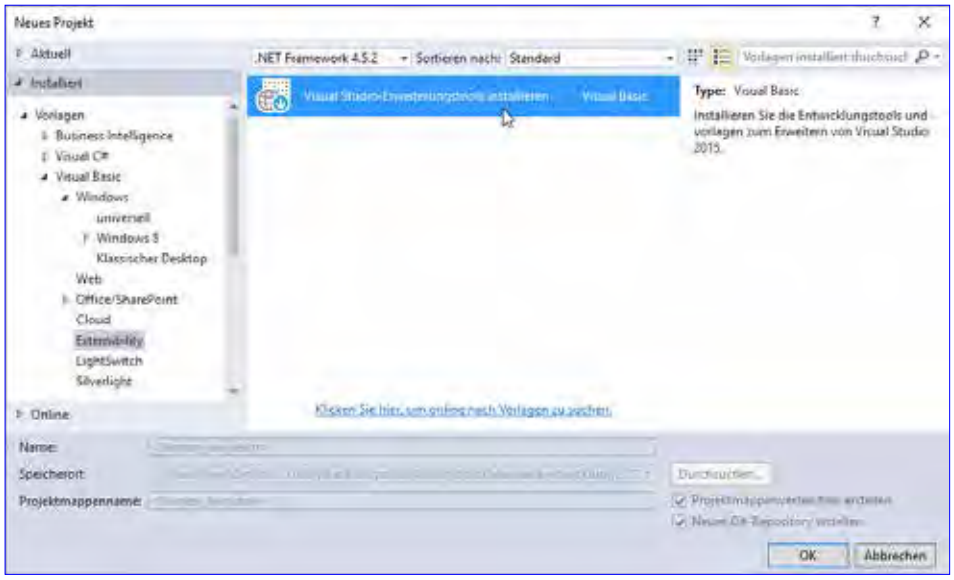


Bild 2: Erweiterungsvorlagen hinzufügen

Vorbereitungen

Wie aber legen wir eine solche Vorlage an und welche Einstellungen sind nötig, damit die Vorlage dann auch beim Anlegen neuer Elemente im Dialog **Neues Element hinzufügen** angezeigt wird? Dazu ist etwas Vorarbeit erforderlich:

- Starten Sie Visual Studio.
- Öffnen Sie den Dialog **Neues Projekt**.
- Wechseln Sie zum Eintrag **Visual C#lExtensibility** oder **Visual BasiclExtensibility**.
- Klicken Sie auf den einzigen Eintrag **Visual Studio Erweiterungstools installieren** (siehe Bild 2).
- Dies startet die Installation (siehe Bild 3).

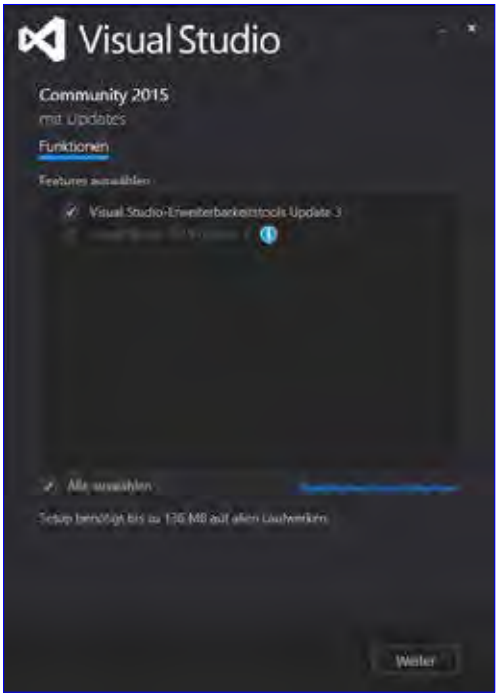


Bild 3: Installation der Erweiterungstools

Wenn Sie, gegebenenfalls nach einem Neustart, nun wieder ein neues Projekt anlegen wollen und dazu den Dialog **Neues Projekt** öffnen, fin-

den Sie im Bereich **Visual C#IExtensibility** oder **Visual BasicIExtensibility** die Einträge aus Bild 4 vor. Uns interessieren hier zunächst die folgenden beiden Einträge:

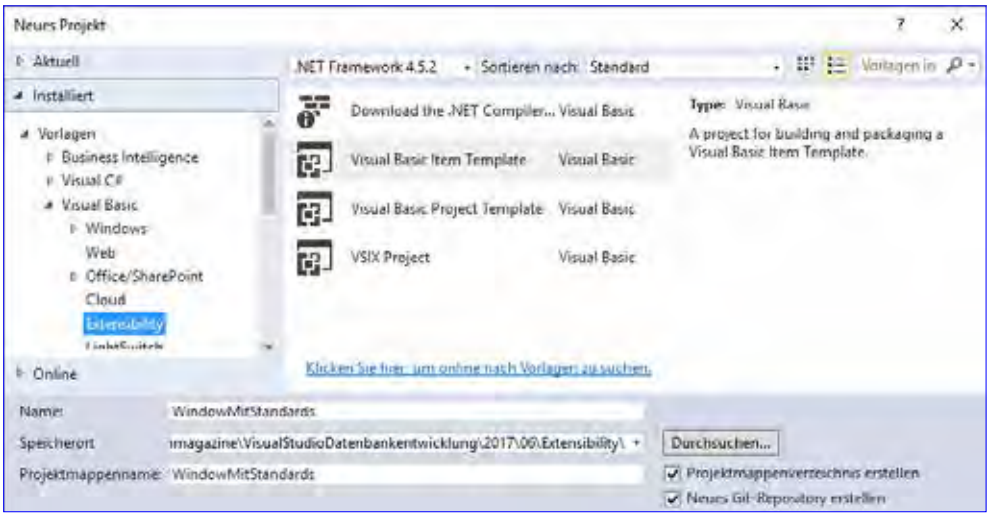


Bild 4: Erweiterungsvorlagen

- **Visual Basic Project Template**: Vorlage für einen Assistenten zum Anlegen neuer Projekte
- **Visual Basic Item Template**: Vorlage für einen Assistenten zum Anlegen eines neuen Elements in Visual Studio, also etwa ein **Class**-, ein **Window**- oder eine **Page**-Element

Wir wollen ein neues Element anlegen, vollständige Projektvorlagen benötigen wir aktuell noch nicht. Also geben wir den Namen des neuen Projekts ein (**WindowMitStandards**), wählen das gewünschte Zielverzeichnis aus und klicken auf die Schaltfläche **OK**.

Das neue Projekt führt sich mit der Anzeige der Datei **ItemWizard.vstemplate** ein (siehe Bild 5). Diese enthält die Konfiguration der frisch erstellten Vorlage. Des Weiteren finden wir im Projektmappen-Explorer einige weitere Dateien:

- **Class.vb**: Bisher einzige Klasse des Projekts.
- **ItemWizard.ico**: Icon-Datei für die Vorlage

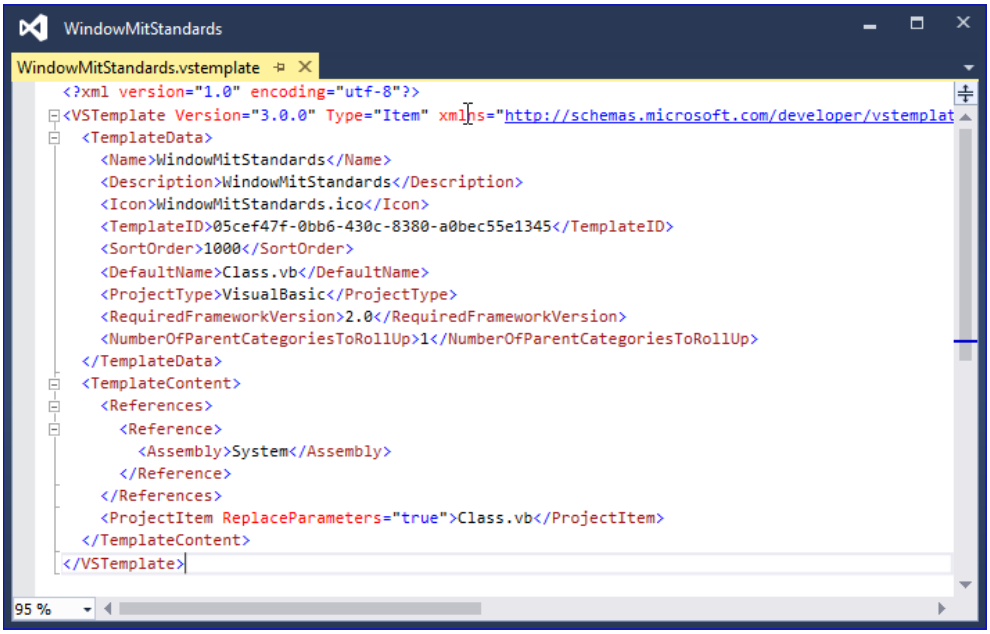


Bild 5: Die Konfigurationsdatei des neuen Projekts

Die Datei **Class.vb** enthält den folgenden Code:

```
Public Class $safeitemname$

End Class
```

Dies lässt bereits erahnen, dass es eine Reihe von Platzhaltern gibt, die wir in die Elemente der Vorlage in-

tegrieren können – wir schauen uns diese weiter unten an. Wozu benötigen wir diese? Nun, wenn Sie beispielsweise eine neue Klasse erstellen, geben Sie im Dialog **Neues Element hinzufügen** ja den Namen des neuen Elements ein. Dieser wird dann für einen der Platzhalter im Code der **.xaml**- oder **.xaml.cs**/**.xaml.vb**-Datei eingetragen.

Vorgehensweise

Nachdem wir das Vorlagen-Projekt erstellt und gesichtet haben, stellt sich die Frage, welche Aufgaben nun vor uns liegen. Diese scheinen wie folgt zu lauten:

- Anlegen des Elements/der Elemente und Versehen des Codes mit Platzhaltern
- Anpassen der Konfigurationsdatei
- Exportieren der Vorlage

Warum »scheinen wie folgt zu lauten«? Ganz einfach: Weil die Konfigurationsdatei beim Exportieren des gewünschten Elements als Vorlage überhaupt nicht berücksichtigt wird. Visual Studio stellt in den nachfolgend beschriebenen Schritten selbst eine Konfigurationsdatei auf Basis der gemachten Angaben zusammen. Sie brauchen sich also nicht die Mühe zu machen, diese direkt in Visual Studio zu manipulieren. Das erledigen wir später, wenn nötig, an anderer Stelle.

Test des Exports

Zu Testzwecken wollen wir einfach die Klasse **Class.vb** übernehmen und dieser als einzige Anpassung einen Kommentar hinzufügen:

```
Public Class $safeitemname$
    'Tolle neue Vorlagenklasse
End Class
```

Den Inhalt der Datei **WindowsMitStandards.vstemplate** wollen wir vorerst beibehalten. Nun folgt schon der entscheidende Schritt: Sie wählen aus dem Menü den Eintrag **Datei|Vorlage exportieren...** aus. Es erscheint der Dialog aus Bild 6, in dem Sie den zweiten Eintrag **Elementvorlage auswählen** und das Projekt **WindowsMitStandards** als Vorlagenquelle beibehalten.

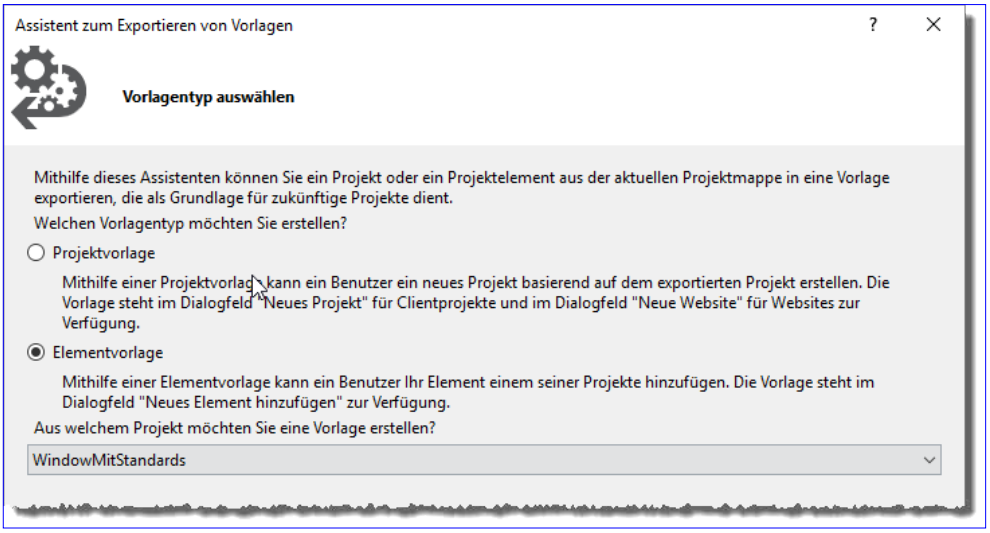


Bild 6: Export als Elementvorlage

Der nächste Schritt des Assistenten fragt, welches Element als Vorlagen-Element verwendet werden soll. Die Auswahl ist einfach, zumal Sie nur einen Eintrag auswählen können – nämlich die Klasse **Class.vb** (siehe Bild 7).

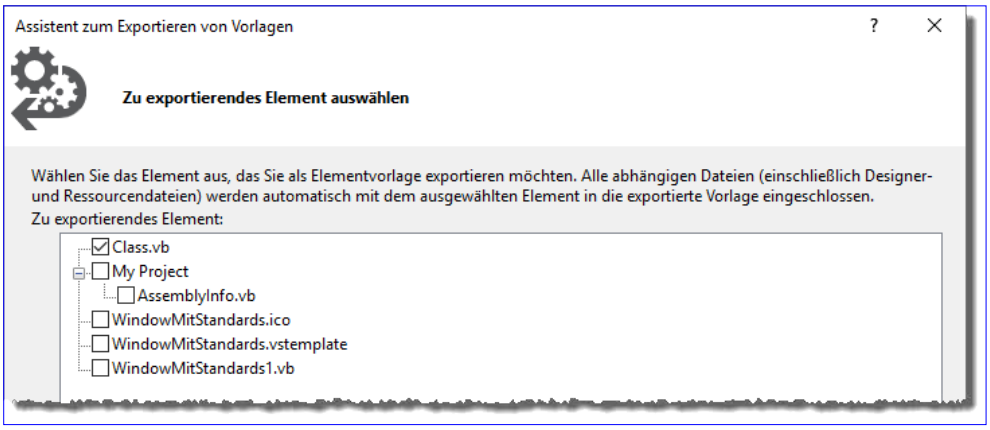


Bild 7: Festlegen des als Vorlage zu verwendenden Elements

Danach erscheint ein Dialog, mit dem Sie die Verweise auswählen können, die gegebenenfalls speziell für das Element benötigt werden. Der Dialog zeigt alle im Vorlagen-Projekt referenzierten Verweise an. Wenn Sie also nachträglich noch Verweise zum Projekt hinzufügen wie etwa den Verweis auf die Bibliothek **Microsoft.Office.Interop.Word**, dann wird auch dieser zur Auswahl angeboten (siehe Bild 8).

Wenn Sie wie in diesem Beispiel einen Verweis auswählen, der nicht mit Visual Studio vorinstalliert wird, weist der Dialog auf eventuelle Probleme hin – in diesem Fall, weil Microsoft Word ja vielleicht gar nicht auf dem Zielrechner der Vorlage installiert ist (bei genauerem Hinsehen liefert der Dialog diesen Hinweis auch für alle standardmäßig enthaltenen Verweise).

Im folgenden und letzten Schritt des Assistenten geben Sie den Namen der anzulegenden Vorlage und einige weitere Informationen an (siehe Bild 9).

Zum Abschluss öffnet Visual Studio dann noch den Ordner, in dem die Vorlagendatei angelegt wurde. Hier können Sie dann auch gleich das entsprechende Verzeichnis entnehmen – in diesem Fall beispielsweise **C:\Users\<Benutzername>\Documents\Visual Studio 2015\My Exported Templates**.

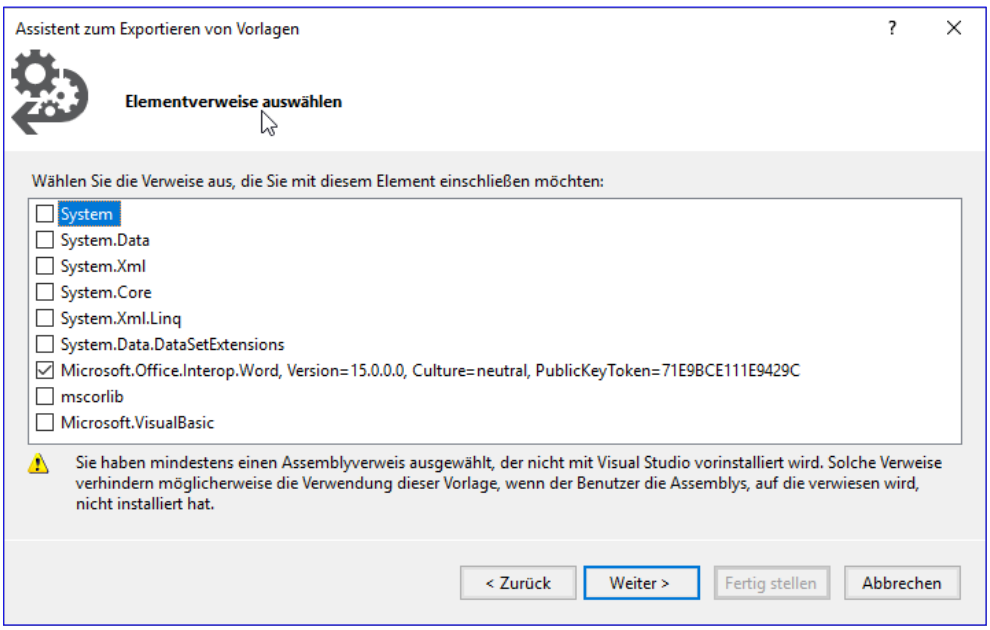


Bild 8: Auswählen der Projektverweise

Neue Vorlage ausprobieren

Schließlich finden Sie, spätestens nach einem

Neustart von Visual Studio, die neue Vorlage im Dialog **Neues Element hinzufügen** vor (siehe Bild 10).

Da wir in der Konfigurationsdatei keine Änderungen vorgenommen haben, verwendet die Vorlage das vom Assistenten mitgelieferte Bild als Icon für die Vorlage. Der Name wurde ebenso übernommen wir der Beschreibungstext.

Klicken Sie nun auf **Hinzufügen**, legt Visual Studio die Klasse unter der angegebenen Bezeichnung an.

Die Klasse wird nun erstellt, allerdings enthält sie nicht den Code, den wir erwartet haben.

Stattdessen wird nicht nur der Platzhalter **$safeitemname$** durch den neuen Klassennamen ersetzt, sondern auch die Bezeichnung **Class**:

```
Public WindowMitStandards1
WindowMitStandards1
    'Tolle neue Vorlagen-
klasse
End WindowMitStandards1
```

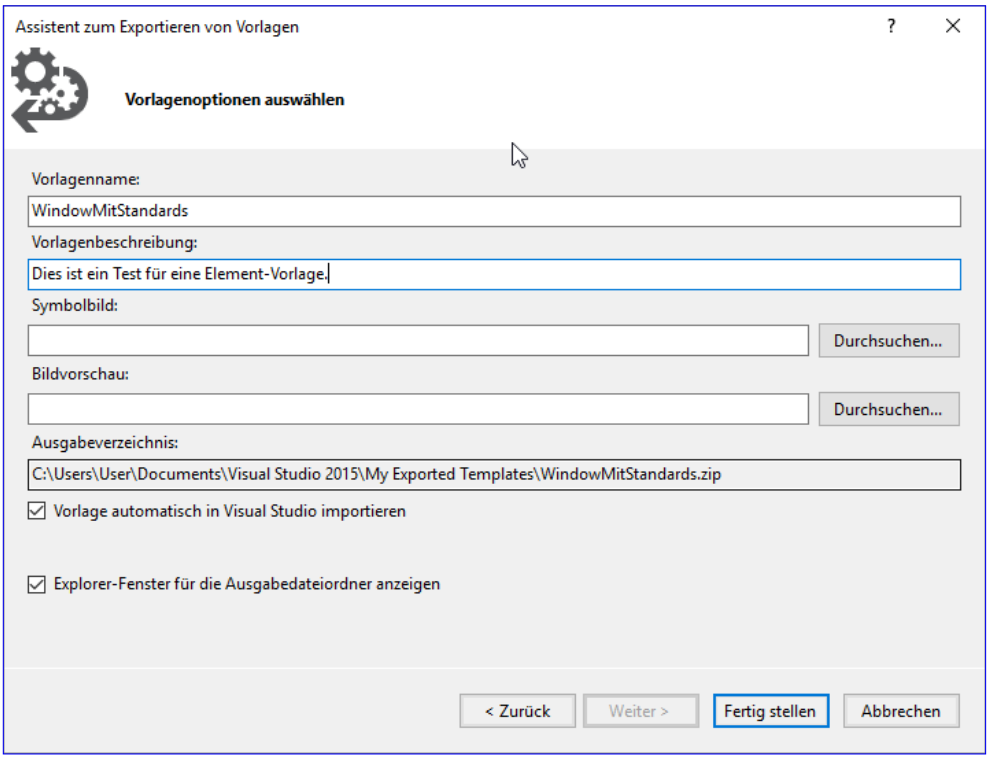


Bild 9: Abschließender Dialog

Bild 10: Die neue Element-Vorlage ist bereits verfügbar.

Warum geschieht dies? Dazu werfen wir einen Blick in die Vorlagendatei mit der Dateiendung **.zip** im Vorlagenverzeichnis. Diese enthält die drei Dateien aus Bild 11.

Wenn wir die Datei **Class.vb** öffnen, erhalten wir folgenden Code:

Grid im Griff

Das Grid-Element ist ein flexibles Steuerelement zum Platzieren der enthaltenen Steuerelemente. Seine Definition erfolgt entweder über XAML oder über den Entwurf der jeweiligen .xaml-Seite. Wie Sie das Grid-Element zur Anordnung von Steuerelementen mit XAML definieren, haben wir uns zum Teil schon im Artikel Steuerelemente anordnen angesehen. Im vorliegenden Artikel zeigen wir nun, wie Sie das Grid schnell mit der Maus so definieren, wie Sie es sonst per XAML tun – und sich damit eine Menge Zeit und Mühe sparen können.

Standardaufgaben vereinfachen

Wenn Sie Steuerelemente auf einem WPF-Fenster anordnen wollen, können Sie diese natürlich einfach so aus der Toolbox auf das Fenster ziehen und mit den absoluten Werten arbeiten, die dann automatisch für Position und Größe der Steuerelemente angelegt werden. Sie können diese allerdings auch in Elementen wie dem **Grid**, dem **StackPanel**, dem **WrapPanel**, dem **DockPanel** oder anderen Elementen platzieren. Wenn Sie die Steuerelemente alle direkt auf der Seite platzieren, wird es kompliziert, wenn es um verschiedene Bildschirmauflösungen oder in der Größe variable Fenster geht. Mit dem **Grid**-Element geht das alles viel besser: Sie können dann die Elemente in das Grid einfügen und durch Einstellen von Eigenschaften dafür sorgen, dass sich die Größe etwa eines Bezeichnungsfeldes, Textfeldes oder einer Schaltfläche in Abhängigkeit der jeweiligen Zelle im Grid anpasst. Erstmal wollen wir uns jedoch anschauen, wie dem Grid Zeilen und Spalten mit der Maus hinzufügen, also durch direkte Manipulation der XAML-Entwurfsansicht.

Grid-Element

Das **Grid**-Element ist ja bei jedem neu angelegten **Window**- oder **Page**-Element automatisch vorhanden und wird im XAML-Code wie folgt definiert:

```
<Grid></Grid>
```

Im Entwurf fällt es auch deshalb kaum auf, weil seine Größe genau mit der des Fensters übereinstimmt. Allein wenn Sie es im Code markieren, wird es durch die Verankerungen sichtbar (siehe Bild 1).

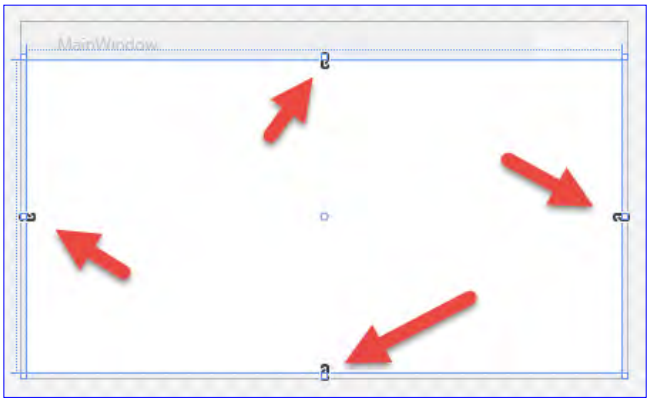


Bild 1: Einfaches Grid, standardmäßig an allen Seiten verankert

Sollten Sie ein Grid einmal versehentlich aus dem übergeordneten Element löschen, können Sie es einfach aus der Toolbox in das Zielelement ziehen – dann wird im Code es allerdings mit absoluten Eigenschaften versehen, etwa so – und sieht im Entwurf wie in Bild 2 aus:

```
<Grid HorizontalAlignment="Left" Height="100" Margin="20,20,0,0" VerticalAlignment="Top" Width="200"/>
```

Wenn das Grid sich über das komplette Fenster erstrecken soll und bei Größenänderungen ebenfalls angepasst werden soll, benötigen Sie all diese Eigenschaften jedoch nicht. Bevor Sie die unnötigen Attribute aus dem Element entfernt haben, haben

Sie jedoch schneller ein neues, leeres Element mit der Definition **<Grid></Grid>** angelegt.

Anlegen von Spalten

Für eine bessere Sichtbarkeit der nachfolgend vorgenommenen Änderungen verwenden wir jedoch weiterhin das mitten im Fenster angelegte **Grid**-Element. Um dem **Grid**-Element eine neue Spalte hinzuzufügen, bewegen Sie den Mauszeiger nach dem Markieren des **Grid**-Elements (was am einfachsten durch Setzen der Einfügemarke in den Code des **Grid**-Elements gelingt) oben an die gestrichelte Linie, bis eine vertikale Linie wie in Bild 3 erscheint.

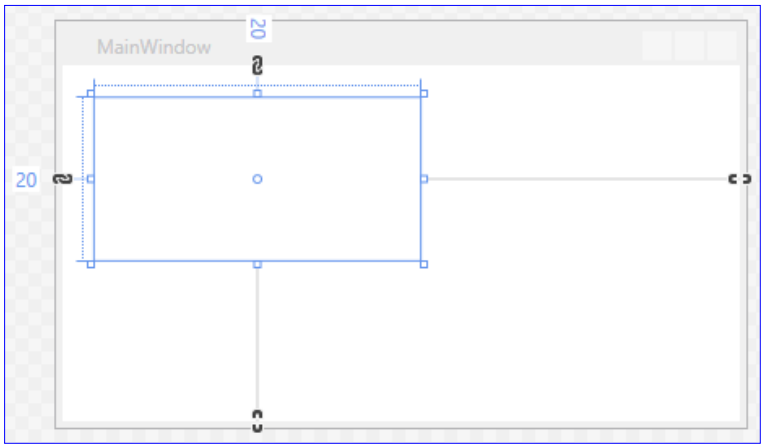


Bild 2: Ein aus der Toolbox hinzugefügtes Grid-Element

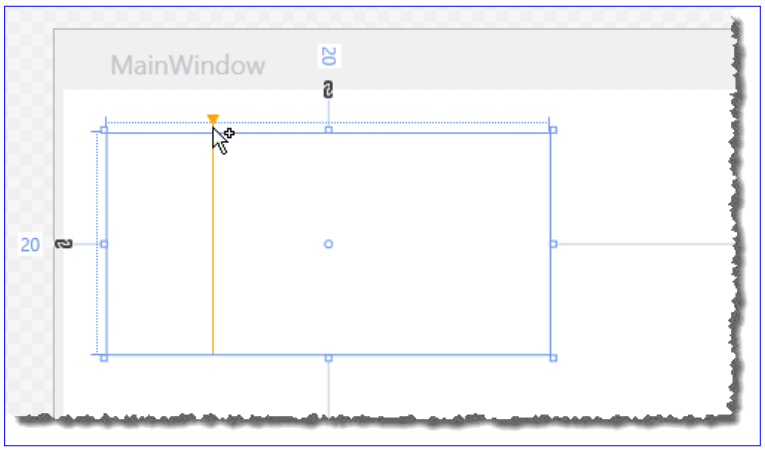


Bild 3: Hinzufügen einer Spalte zum **Grid**-Element

Bewegen Sie die Maus an die gewünschte Stelle und klicken Sie, wenn Sie diese erreicht haben. Die neue Spalte wird dann wie in Bild 4 angelegt. Der XAML-Code wird dadurch merklich erweitert, nämlich um die Definition der entsprechenden Spalten. Durch das Hinzufügen einer Spalte haben Sie nun zwei Spalten, denn eine Spalte war ja implizit bereits vorher vorhanden. Nun haben wir zwei durch **ColumnDefinition**-Elemente definierte Spalten, für die Visual Studio gleich die entsprechenden Breiten eingetragen hat:

```
<Grid ...>
    <Grid.ColumnDefinitions>
        <ColumnDefinition Width="41*"/>
        <ColumnDefinition Width="159*"/>
    </Grid.ColumnDefinitions>
</Grid>
```

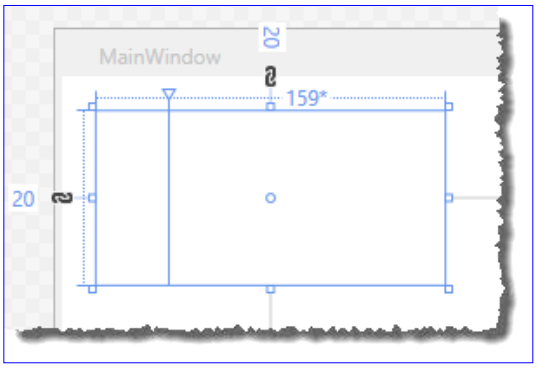


Bild 4: **Grid**-Element mit der neuen Spalte

Wenn Sie weitere vertikale Linien auf die oben gezeigte Art hinzufügen, werden die Elemente unter **Grid.ColumnDefinitions** entsprechend erweitert. Was bedeutet **41*** und **159*** doch gleich? Das ist das Verhältnis, in dem die Breiten der Spalten zueinanderstehen. Die Gesamtbreite entspricht hier 200*, also nimmt die linke Spalte 41/200 und die rechte 159/200 der Breite de **Grid**-Elements ein. Wenn Sie dies einfach anpassen wollen, bewegen Sie den Mauszeiger oben auf das nach unten zeigende Dreieck zwischen den beiden Spalten und ziehen

Sie dieses wie in Bild 5 nach links oder rechts. Halten Sie dabei die **Umschalt**-Taste gedrückt, ändert sich auch gleich noch die Größe des dem Grid übergeordneten Elements, hier dem **Window**-Element, mit.

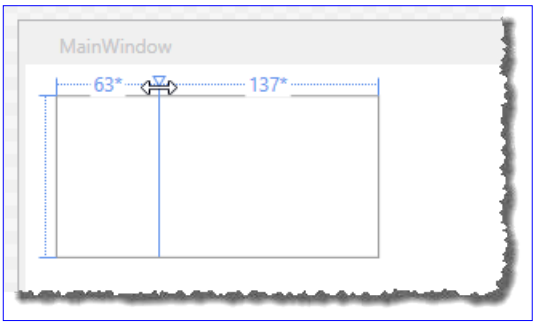


Bild 5: Ändern des Spaltenbreiten-Verhältnisses

Neben der hier beschriebenen Syntax mit der Zahl und dem folgenden Sternchen (*) wie **50*** gibt es natürlich noch andere Werte, die Sie gelegentlich verwenden wollen – zum Beispiel das Sternchen ohne führende Zahl oder der Wert **Auto**. Diese können Sie ebenfalls über die Entwurfsansicht manipulieren. Dazu fahren Sie mit der Maus wie oben beschrieben so in den oberen Bereich einer Spalte, als ob Sie eine neue Spalte anlegen wollen. Es erscheint dann eine Art Kombinationsfeld, das nach dem Anklicken der Schaltfläche am rechten Rand die Optionen wie in Bild 6 anzeigt.

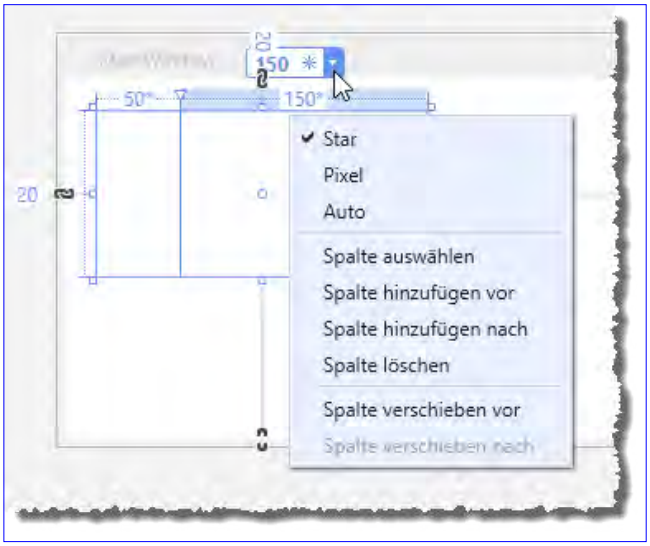


Bild 6: Spalten-Optionen

- **Star**: Entspricht der Standardeinstellung.
- **Pixel**: Trägt die aktuelle Breite in Pixeln für die Spaltenbreite ein.
- **Auto**: Trägt den Wert Auto für die Eigenschaft **Width** ein. Wenn keine Steuerelemente enthalten sind, führt dies zur Anzeige der Spalte mit der Breite **0**.

Die übrigen Einträge erlauben das Auswählen der gewählten Spalte, das Hinzufügen einer Spalte vor oder nach der gewählten Spalte oder das Löschen der aktuellen Spalte. Der Eintrag **Spalte verschieben vor** tauscht die Position der Spalte mit der links davon befindlichen Spalte, der Eintrag **Spalte verschieben nach** tauscht mit der rechts davon liegenden Spalte.

Sie können die Maus allerdings auch über die Zahl oder das Symbol im Kombinationsfeld fahren und dann auf eines der beiden Elemente klicken. Mit einem Klick auf die Zahl können Sie den Zahlenwert ändern, mit einem Klick auf das Symbol wechseln Sie durch die unterschiedlichen Werte, nämlich Sternchen (*), Pixelgröße (dargestellt durch ein Schloss) und **Auto** (siehe Bild 7).

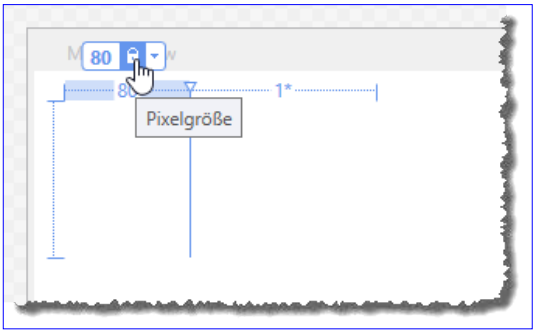


Bild 7: Direktes Manipulieren der Breite im Entwurf

Zeilen hinzufügen

Das Hinzufügen von Zeilen gelingt analog zu den Spalten – Sie müssen nur an die entsprechende Stelle an der vertikalen gestrichelten Linie des **Grid**-Elements klicken. Dadurch fügen Sie dann zunächst eine neue Zeile hinzu, was wie in Bild 8 aussieht. Der XAML-Code des **Grid**-Elements wird dadurch wiederum stark erweitert, und zwar um ein **Grid.RowDefinitions**-Element und für jede Zeile ein **RowDefinition**-Element:

Visual Basic: Bedingungen und Schleifen

Wer bisher mit VBA oder C# gearbeitet hat und zu VB wechseln möchte, sieht sich bei der Entwicklung von WPF-Anwendungen einigen Änderungen gegenüber. Dieser Artikel liefert die Grundlagen zur Programmierung von Bedingungen wie If...Then oder Select Case und von Schleifen wie For...Next, For...Each oder Do While. Zum Experimentieren mit den Beispielen nutzen wir das Tool LINQPad 5.

Wer von VBA kommt, wird wenig Probleme haben, sich in Visual Basic für .NET einzuarbeiten. Wer nur C# kennt oder sich daran gewöhnt hat, muss sich erst ein wenig einarbeiten. Hier kommen nämlich die wichtigsten Konstrukte bei der Programmierung von Methoden und Funktionen: Bedingungen und Schleifen. Die Beispiele dieses Artikels haben wir der Einfachheit halber wieder im Tool LINQPad5 ausprobiert, das wir im Artikel **LINQPad: LINQ, C# und VB einfach ausprobieren** bereits vorgestellt haben.

Einfaches If...Then

Eine einfache **If...Then**-Bedingung prüft, ob der angegebene Ausdruck wahr ist und führt dann die zwischen der **If**- und der **End If**-Zeile enthaltenen Anweisungen aus:

```
Dim intZahl As Integer
intZahl = 10
If intZahl > 5 Then
    Debug.WriteLine("intZahl ist größer als 5")
End If
```

Von C# kommend ist das eine Umstellung, denn auch hier fallen die geschweiften Klammern weg und die Semikola am Ende der Befehlszeilen. Auch ist es hier nicht mehr möglich, eine einfache Zeile einfach ohne geschweifte Klammern hinter der **If**-Zeile einzufügen. Es gibt aber auch unter VB eine vereinfachte Schreibweise für einfache Anweisungen. Hier lassen Sie dann einfach die Zeile **End If** weg und fügen die auszuführende Zeile einfach an die Bedingung an:

```
If intZahl > 5 Then Debug.WriteLine("intZahl ist größer als 5")
```

If...Then...Else-Bedingung

Nehmen wir noch einen **Else**-Zweig hinzu, um Anweisungen anzugeben, die bei Nichterfüllung der Bedingung ausgeführt werden sollen. Dann sieht die Bedingung wie folgt aus:

```
If intZahl > 5 Then
    Debug.WriteLine("intZahl ist größer als 5")
Else
    Debug.WriteLine("intZahl ist nicht größer als 5")
End If
```

Wenn der erste Teil der Bedingung hier nicht eintritt, wird auf jeden Fall der zweite Teil hinter dem Schlüsselwort **Else** ausgeführt. Sie können statt **Else** auch das Schlüsselwort **ElseIf** verwenden und eine zweite Bedingung angeben. Im folgenden Beispiel müssen wir allerdings beachten, dass im Falle von **intZahl = 5** keiner der beiden Zweige ausgeführt wird:

```
If intZahl > 5 Then
    Debug.WriteLine("intZahl ist größer als 5")
ElseIf intZahl < 5 Then
    Debug.WriteLine("intZahl ist nicht größer als 5")
End If
```

Für diesen Fall können wir aber auch noch einen finalen **Else**-Zweig anhängen, der auf jeden Fall ausgeführt wird. Der **Else**-Zweig kann allerdings nur als letzte Bedingung verwendet werden:

```
If intZahl > 5 Then
    Debug.WriteLine("intZahl ist größer als 5")
ElseIf intZahl < 5 Then
    Debug.WriteLine("intZahl ist nicht größer als 5")
Else
    Debug.WriteLine("Keine der angegebenen Bedinungen traf zu.")
End If
```

Sie können mehrere **ElseIf**-Zweige einsetzen. Das ist ein großer Unterschied zu C#, denn dort gibt es gar kein **ElseIf** (nur eine vereinfachte Schreibweise für verschachtelte **If**-Bedingungen mit **Else If**). Hier müssen Sie für solche Fälle mit verschachtelten Bedingungen arbeiten, wenn Sie **If...Then** nutzen wollen.

Verschachtelte If...Then-Bedingung

If...Then-Bedingungen lassen sich ineinander verschachteln:

```
If intZahl > 5 Then
    Debug.WriteLine("intZahl ist größer als 5")
ElseIf
    If intZahl > 3 Then
        Debug.WriteLine("intZahl ist größer als 3 und kleiner als 5")
    Else
        Debug.WriteLine("intZahl ist kleiner gleich 3")
    End If
End If
```

Verwendung von If...Then in einer Zeile mit IIf

Die von VBA bekannte Funktion **IIf** können Sie unter C# nicht verwenden, es gibt dort eine alternative Schreibweise mit gleicher Funktion. Unter VB jedoch steht **IIf** wie unter VBA zur Verfügung. Dabei erwartet die Funktion drei Parameter:

- Bedingung
- Wert, wenn die Bedingung wahr ist
- Wert, wenn die Bedingung falsch ist

Die obige **If...Then...Else**-Bedingung würde in einer Zeile wie folgt aussehen:

```
IIf(intZahl>5, "Zahl ist größer als 5", "Zahl ist kleiner gleich 5")
```

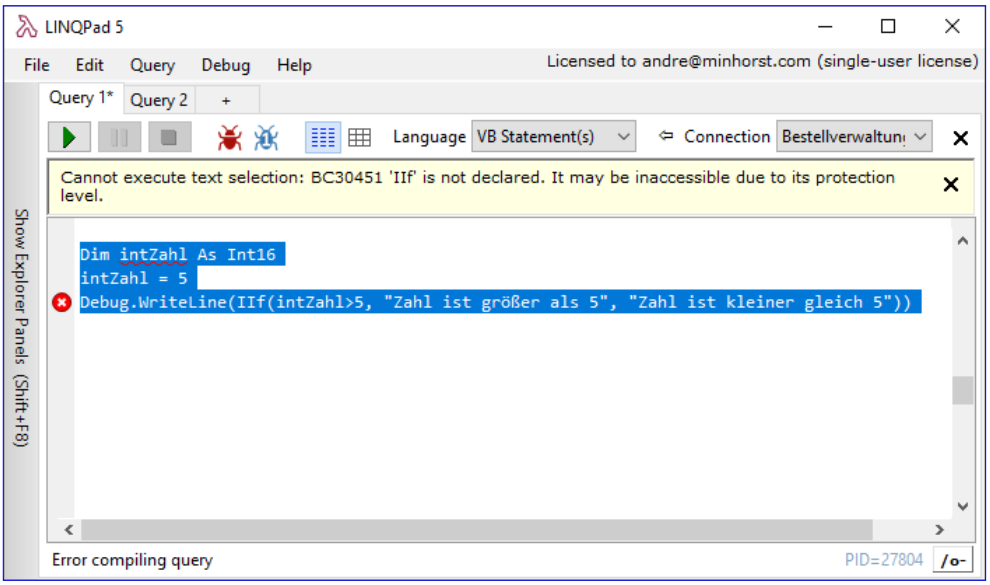


Bild 1: Fehler beim Ausführen der Funktion **IIf**

Hinweis: Wenn Sie **IIf** und anderen Visual Basic-Funktionen in LINQPad testen wollen, erhalten Sie normalerweise den Fehler aus Bild 1.

Um dies zu verhindern, müssen Sie den Namespace **Microsoft.VisualBasic** hinzufügen.

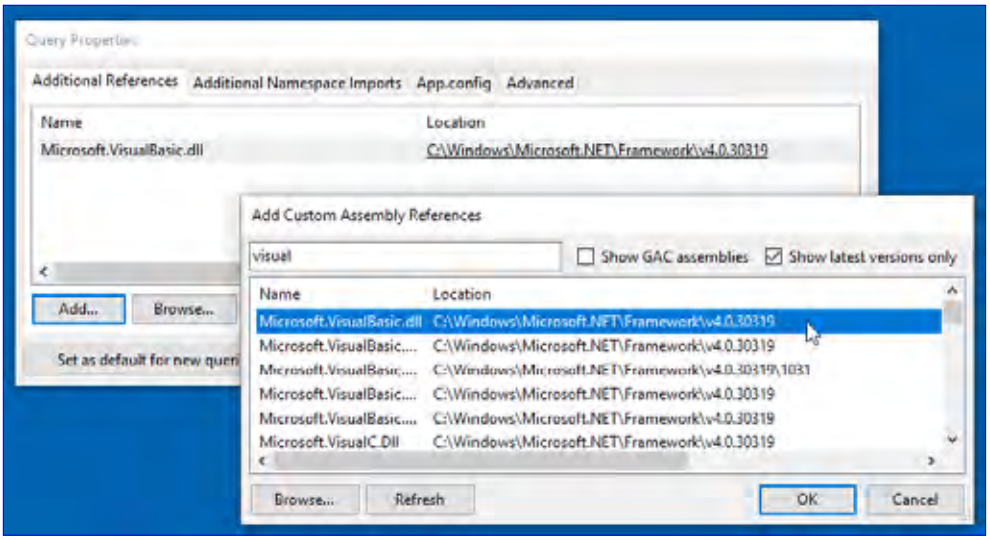


Bild 2: Hinzufügen der Referenz auf die Visual Basic-Bibliothek

Das erledigen Sie über einen Dialog, den Sie mit dem Menübefehl **Query|Namespace Imports** öffnen. Der Dialog heißt **Query Properties**. Im Bereich **Additional Reference** klicken Sie auf die Schaltfläche **Add**, was den Dialog **Add Custom Assembly Reference** öffnet. Hier geben Sie im Suchfeld **VisualBasic** ein und fügen den Eintrag wie in Bild 2 mit einem Klick auf die **OK**-Schaltfläche hinzu.

Nun fehlt nur noch ein Schritt: Zurück im Dialog **Query Properties** wechseln Sie zum Bereich **Additional Namespace Imports** und geben dort wie in Bild 3 gezeigt **Microsoft.VisualBasic** ein. Da wir zuvor einen Verweis auf die entsprechende Bibliothek hinzugefügt haben, können Sie in der Liste oben rechts den Eintrag **Microsoft.VisualBasic.dll** auswählen.

Darunter erscheinen dann alle verfügbaren Namespaces, von denen Sie **Microsoft.VisualBasic** auswählen und per Klick auf die Schaltfläche **< Add Selected Namespaces** zur Liste im rechten Bereich hinzufügen.

Danach können Sie **IIf** nicht nur ohne Fehlermeldung nutzen, sondern **IIf** und die übrigen Elemente des Namespaces werden per IntelliSense angezeigt.

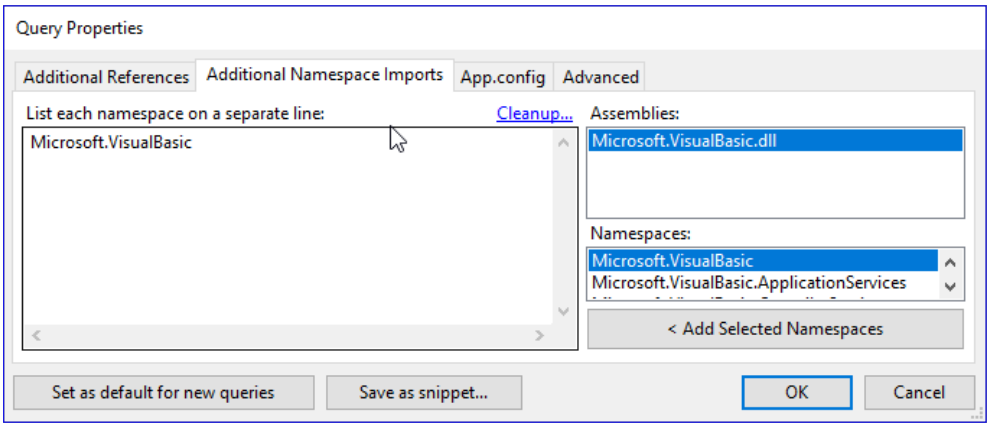


Bild 3: Hinzufügen des Namespaces **Microsoft.VisualBasic**

Mehrere Bedingungen komfortabel mit Select Case prüfen

Wenn Sie unter VBA mehrere Bedingungen prüfen wollen, können Sie das durchaus mit **If...ElseIf...Else...End If** erledigen. Allerdings gibt es auch noch die **Select Case**-Bedingung, mit der Sie Bedingungen mit mehreren Zweigen einfacher abbilden können. **Select Case** ist genauso wie unter VBA aufgebaut. Unter C# setzt man zu diesem Zweck die **switch**-Bedingung ein. **Select Case** funktioniert so:

```
Dim intZahl As Integer
intZahl = 5
Select Case intZahl
    Case >5
        Debug.WriteLine ("Zahl ist größer als 5.")
    Case <5
        Debug.WriteLine ("Zahl ist kleiner als 5.")
    Case Else
        Debug.WriteLine ("Zahl ist gleich 5.")
End Select
```

Wir geben also den ersten Teil des Vergleichsausdrucks hinter der **Select Case**-Zeile an. Der Rest der jeweiligen Vergleichsausdrücke landet jeweils hinter den **Case**-Zweiten. Es gibt einfache **Case**-Zeilen und maximal einen **Case Else**-Zweig, der alle Fälle auffängt, die in den vorherigen Fällen nicht berücksichtigt wurden.

Sie müssen auch nicht wie bei der **switch**-Bedingung unter C# für jede Bedingung eine Anweisung einfügen, welche die Bedingung verlässt, wenn einer der Zweige angesteuert wird. Wenn der erste Case-Zweig angesteuert wurde, werden die übrigen automatisch nicht mehr ausgeführt.

Bei der **switch**-Bedingung unter C# konnten Sie auch einen **case**-Zweig leerlassen. In diesem Fall wurden, wenn dieser **case**-Zweig angesteuert wurde, die Anweisungen des folgenden **case**-Zweiges ausgeführt. Das ist bei **Select Case** nicht der Fall. Enthält ein **Case**-Zweig keine Anweisungen, geschieht auch nichts.

Select Case-Anweisungen, die nur eine Anweisung in einem **Case**-Zweig enthalten, können Sie auch wie folgt vereinfacht schreiben:

```
Dim intZahl As Integer
intZahl = 5
Select Case intZahl
        Case > 5: Debug.WriteLine("Zahl ist größer als 5.")
        Case < 5: Debug.WriteLine("Zahl ist kleiner als 5.")
        Case Else: Debug.WriteLine("Zahl ist gleich 5.")
End Select
```

Grundsätzlich können Sie mehrere Anweisungen durch einen Doppelpunkt voneinander getrennt in einer Zeile aufführen.

Mögliche Ausdrücke für die Case-Zweige

Sie können dem **Case**-Zweig flexiblere Vergleichsausdrücke hinzufügen wie einer **If...Then**-Bedingung.

Wenn Sie beispielsweise in einer **If...Then**-Bedingung die gleiche Variable mit drei verschiedenen Werten vergleichen wollen, müssen Sie dies wie folgt formulieren:

```
Dim intZahl As Integer
intZahl = 5
If intZahl = 1 Or intZahl = 3 Or intZahl = 5 Then
        Debug.WriteLine("intZahl ist 1, 3 oder 5")
End If
```

Hier arbeiten wir ja noch mit einem einfachen Ausdruck. Dieser kann aber auch komplizierter ausfallen, wodurch sich die Schreibarbeit und somit auch die Fehleranfälligkeit massiv erhöhen. Mit **Select Case** geht das viel eleganter:

```
Dim intZahl As Integer
intZahl = 5
Select Case intZahl
    Case 1, 3, 5
        Debug.WriteLine("intZahl ist 1, 3 oder 5")
End Select
```

Das hier hingegen würde nicht funktionieren:

```
Select Case intZahl
    Case 1 Or 3 Or 5
        Debug.WriteLine("intZahl ist 1, 3 oder 5")
End Select
```

Das Ergebnis lautet nämlich 7. Was, 7? Nun: **Or** macht einen binären Vergleich, addiert also die binären Werte: **1** ist **001**, **3** ist **011**, **5** ist **101**. Der **Or**-Vergleich liefert dann **111**, was dezimal **7** entspricht.

E-Mails mit Outlook verschicken

Wer Anwendungen mit Visual Studio programmiert, die Daten wie etwa die von Kunden verwaltet, kommt früher oder später nicht um eine Funktion zum Versenden von E-Mails herum. Da gibt es nun zwei Möglichkeiten: Sie verwenden eine eigene SMTP-Klasse, um die E-Mails zu versenden. Das klappt mit .NET-Projekten viel einfacher als etwa unter Access, weil es hier schon fertige Klassen für diesen Anwendungszweck gibt. Allerdings werden die gesendeten Mails dann nicht in Outlook im Ordner »Gesendete Elemente« gespeichert. Deshalb schauen wir uns in diesem Artikel den Versand von Mails per Outlook an. In einem anderen Artikel gehen wir dann auf die Verwendung einer SMTP-Klasse ein.

Beispielanwendung

Wir wollen als Beispiel für diesen Artikel eine kleine WPF-Anwendung programmieren, mit der Sie die Daten des E-Mail-Empfängers sowie Betreff, Inhalt und weitere Informationen eingeben können. Eine Schaltfläche soll dann die eingegebenen Daten an eine Outlook-Instanz schicken und die E-Mail versenden.

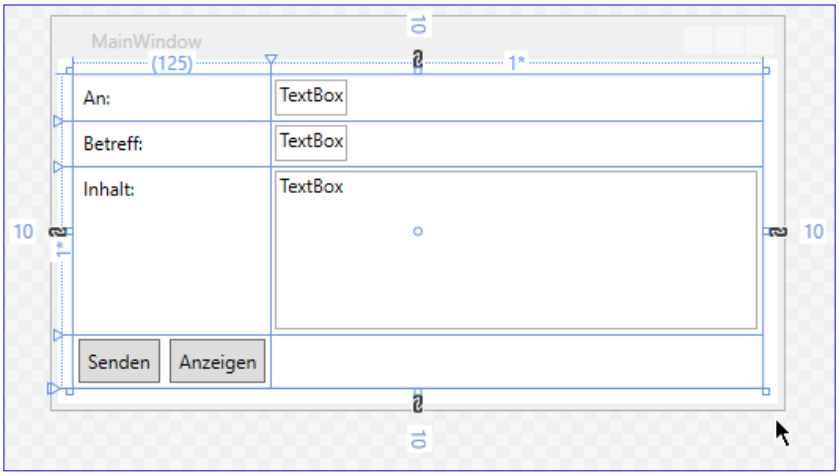


Bild 1: Einfaches Formular zum Versenden von E-Mails

Also legen Sie zunächst eine WPF-Anwendung in der von Ihnen bevorzugten Sprache an (also etwa VB oder C#). Das Fenster **MainWindow.xaml** ist dann auch gleich das Element, das wir mit den benötigten Steuerelementen versehen wollen. Dieses gestalten wir wie in Bild 1.

Der dazu notwendige XAML-Code sieht gekürzt so aus (es gibt noch einige Steuerelement-spezifische Eigenschaftsdefinitionen, die wir nicht abbilden):

```
<Window x:Class="MainWindow" ... Title="MainWindow" Height="250" Width="466">
    <Grid HorizontalAlignment="Stretch" Margin="10" VerticalAlignment="Stretch" ClipToBounds="true" >
        <Grid.RowDefinitions>
            <RowDefinition Height="Auto"/>
            <RowDefinition Height="Auto"/>
            <RowDefinition/>
            <RowDefinition Height="Auto"/>
            <RowDefinition Height="Auto"/>
        </Grid.RowDefinitions>
```

```
        <Grid.ColumnDefinitions>
            <ColumnDefinition Width="Auto"/>
            <ColumnDefinition Width="*"/>
        </Grid.ColumnDefinitions>
        <Label x:Name="label1" Content="An:" />
        <Label x:Name="label2" Content="Betreff:" Grid.Row="1"/>
        <Label x:Name="label3" Content="Inhalt:" Grid.Row="2"/>
        <TextBox x:Name="txtAn" Grid.Column="1" HorizontalAlignment="Stretch"/>
        <TextBox x:Name="txtBetreff" Grid.Row="1" Grid.Column="1" HorizontalAlignment="Stretch"/>
        <TextBox x:Name="txtInhalt" Grid.Row="2" TextWrapping="Wrap" Grid.Column="1" VerticalAlignment="Stretch"
            HorizontalAlignment="Stretch" Height="Auto" AcceptsReturn="True"/>
        <StackPanel Orientation="Horizontal" Grid.Row="3">
            <Button x:Name="btnSenden">Senden</Button>
            <Button x:Name="btnAnzeigen">Anzeigen</Button>
        </StackPanel>
    </Grid>
</Window>
```

Dadurch, dass wir für die dritte Zeile den Wert der Eigenschaften **Height** und **Width** auf **Auto** eingestellt haben, wird diese Zeile gemeinsam mit dem Fenster vergrößert. Damit wir im Textfeld **txtInhalt** mehrzeilige Texte eingeben können, haben wir die Eigenschaft **TextWrapping** auf **Wrap** und **AcceptsReturn** auf **True** eingestellt.

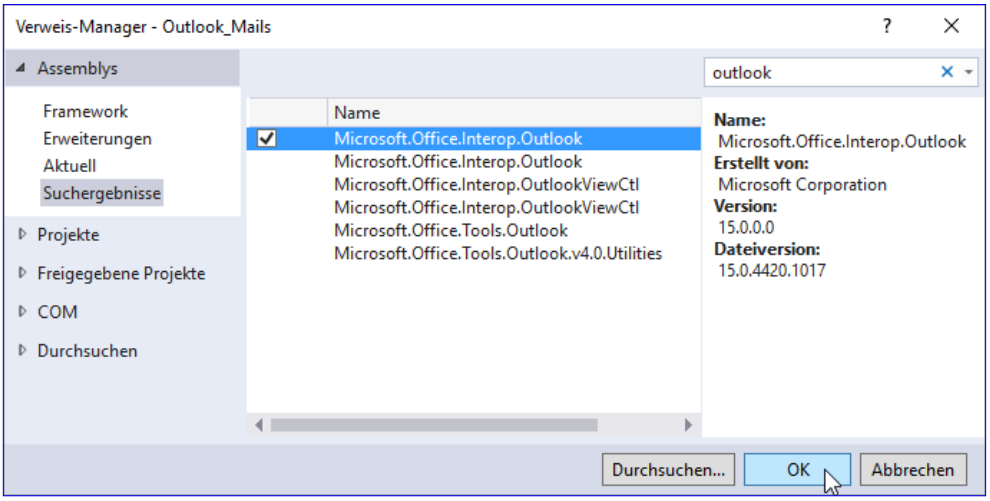


Bild 2: Erweitern der Verweise des Projekts um die Outlook-Bibliothek

Verweis auf Outlook hinzufügen

Damit wir Outlook von einer WPF-Anwendung aus programmieren können, fügen wir den entsprechenden Verweis auf die Outlook-Bibliothek **Microsoft.Office.Interop.Outlook** hinzu. Dazu öffnen Sie den **Verweise-Manager** über den Menüeintrag **Projekt|Verweis hinzufügen...** und suchen im nun erscheinenden Dialog nach dem Wort **Outlook**.

Selektieren Sie den Eintrag **Microsoft.Office.Outlook.Interop** (siehe Bild 2) und schließen Sie den Dialog mit **OK**. Sollten Sie, wie im obigen Dialog der Fall, mehrere gleichnamige Einträge vorfinden, handelt es sich um verschiedene Versionen. Wählen Sie dann die aktuellere Version aus.

Dann statten wir die Schaltfläche **cmdSenden** mit der Ereigniseigenschaft **Click** aus. Nach der Eingabe des Gleichheitszeichens können Sie mit der Tabulator-Taste automatisch die entsprechende Ereignismethode anlegen, indem Sie aus der dann erscheinenden Auswahlliste den Eintrag **<Neuer Ereignishandler>** auswählen:

```
<Button x:Name="btnSenden" Click="btnSenden_Click">Senden</Button>
```

In der Code behind-Klasse erscheint dann die folgende Methode:

```
Private Sub btnSenden_Click(sender As Object, e As RoutedEventArgs)

End Sub
```

In der Klasse fügen wir nun zunächst eine **Imports**-Anweisung hinzu, mit der wir den Outlook-Namespace verfügbar machen:

```
Imports Outlook = Microsoft.Office.Interop.Outlook
```

Damit greifen wir über das Schlüsselwort **Outlook** auf Elemente dieses Namespaces zu.

Textfelder mit Beispiel füllen

Bevor wir uns auf die Outlook-Programmierung stürzen, legen wir noch eine Ereignismethode für das Ereignis **Loaded** des **Window**-Elements an. Diese soll die drei Textfelder mit Beispieltexten füllen, damit das Fenster beim Öffnen gleich wie in Bild 3 aussieht:

```
Private Sub Window_Loaded(sender As Object, e As RoutedEventArgs)
    txtAn.Text = "andre@minhorst.com"
    txtBetreff.Text = "Testmail"
    txtInhalt.Text = "Hallo Herr Minhorst," & vbCrLf _
        & vbCrLf _
        & "dies ist eine Testmail." & vbCrLf _
        & vbCrLf _
        & "Mit freundlichen Grüßen" & vbCrLf _
        & "André Minhorst"
End Sub
```

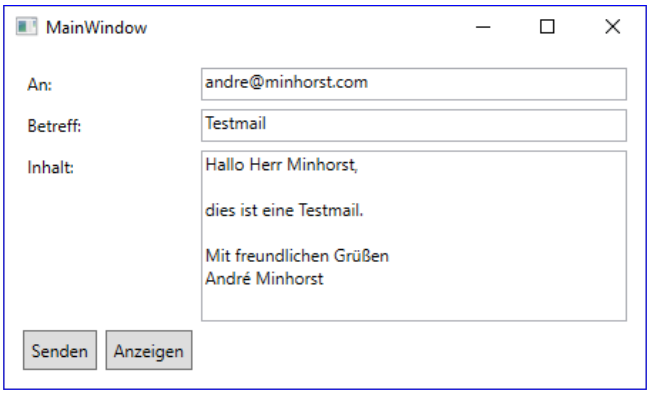


Bild 3: Beispielfenster zum Versenden von E-Mails

Outlook-Mail erzeugen

Danach kümmern wir uns um den Code, der die Mail mit den angegebenen Daten versenden soll. Dazu füllen wir die bereits angelegte Ereignismethode **btnSenden_Click** wie folgt:

```
Private Sub btnSenden_Click(sender As Object, e As RoutedEventArgs)
    Dim objOutlook As Outlook.Application
```

```
    Dim objMailItem As Outlook.MailItem
    objOutlook = New Outlook.Application
    objMailItem = objOutlook.CreateItem(Outlook.OlItemType.olMailItem)
    With objMailItem
        .To = txtAn.Text
        .Subject = txtBetreff.Text
        .Body = txtInhalt.Text
        .Send()
    End With
End Sub
```

Wir erstellen also ein Objekt des Typs **Microsoft.Office.Interop.Outlook.Application**, den wir hier aufgrund unserer **Imports**-Anweisung vereinfacht mit **Outlook.Application** referenzieren können, und speichern einen Verweis darauf in der Variablen **objOutlook**. Die zweite Variable namens **objMailItem** mit dem Datentyp **MailItem** nutzen wir, um ein mit der **CreateItem**-Methode von **objOutlook** erzeugtes **MailItem**-Objekt zu referenzieren.

Dies dient dem Zweck, seine Eigenschaften **To**, **Subject** und **Body** mit den Inhalten der drei Textfelder **txtAn**, **txtBetreff** und **txtInhalt** zu füllen. Die letzte Anweisung heißt **Send** und sorgt dafür, dass die Mail direkt verschickt wird.

Wenn Sie die Anwendung nun starten und die Mail mit einem Klick auf die Schaltfläche **cmdSenden** verschicken, können Sie diese anschließend im Ordner **Gesendete Objekte** von Outlook einsehen.

E-Mail erst anzeigen

Etwas transparenter ist das Ganze natürlich noch, wenn Sie die E-Mail betrachten können, bevor Sie diese durch einen Klick auf die dafür vorgesehene Schaltfläche absenden. Dazu legen wir für die Schaltfläche **cmdAnzeigen** ebenfalls die **Click**-Eigenschaft an:

```
<Button x:Name="btnAnzeigen" Click="btnAnzeigen_Click">Anzeigen</Button>
```

Den Code dieser Methode brauchen wir gegenüber der zuvor beschriebenen nur leicht zu variieren:

```
Private Sub btnAnzeigen_Click(sender As Object, e As RoutedEventArgs)
    ...
    With objMailItem
        ...
        .Display()
    End With
End Sub
```

Hier rufen wir nämlich statt der **Send**-Methode die **Display**-Methode auf. Dies zeigt beim Ausprobieren die Mail aus Bild 4 an. Nach dem Betrachten schickt man die E-Mail dann mit einem Klick auf die **Senden**-Schaltfläche ab.

Absender von Outlook-Mails einstellen

Im Artikel »E-Mails mit Outlook verschicken« haben wir gezeigt, wie Sie von einer .NET-Anwendung aus die Daten für eine E-Mail zusammenstellen und diese dann per Outlook verschicken können – unter anderem mit Empfänger, CC, BCC, Anlagen oder Priorität. Was noch fehlt, ist die Absenderadresse. Diese braucht man grundsätzlich nicht explizit einzustellen, denn Outlook verwendet dann die Adresse des Standardkontos von Outlook (vorausgesetzt, Sie haben mehrere Konten). Wenn Sie jedoch eine Mail mit einem anderen Konto als dem Standardkonto verschicken wollen, wird es kompliziert. Die Lösung für diese Aufgabe finden Sie im vorliegenden Artikel.

Verschiedene Outlook-Konten

Wenn Sie wie im **Artikel E-Mails mit Outlook verschicken** ein **MailItem**-Objekt erzeugen und seine Eigenschaften wie **To**, **CC**, **BCC**, **Subject** oder **Body** füllen, werden Sie vergeblich ein Feld etwa namens **From** suchen. Dieses Feld gibt es nämlich nicht – Sie können also nicht einfach irgendeine Absenderadresse zu Ihrer E-Mail hinzufügen.

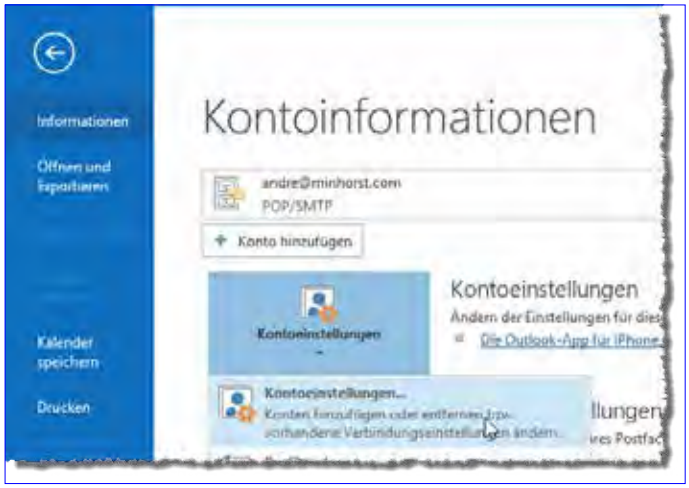


Bild 1: Anzeigen der Kontoeigenschaften

Woher weiß Outlook dann aber, welche Absenderadresse überhaupt zu verwenden ist? Ganz einfach: Es gibt in Outlook ein oder mehrere Konten, von denen eines als Standard festgelegt ist. Wann immer Sie direkt von Outlook aus eine neue E-Mail erstellen, wird die zum Standardkonto gehörende E-Mail-Adresse als Absenderadresse eingestellt. Wo finden Sie diese Konten und Einstellungen? Dazu öffnen Sie Outlook, klicken auf den Ribbon-Tab **Datei** und dann im Bereich **Informationen** auf die Schaltfläche **Kontoeinstellungen|Kontoeinstellungen...** (siehe Bild 1).

Hier finden Sie dann wie in Bild 2 die angelegten Outlook-Konten vor. Hier können Sie beispielsweise doppelt auf einen der Einträge klicken, um Eigenschaften wie den SMTP-Server oder die E-Mail-Adresse anzupassen oder ein anderes als das aktuelle Konto als Standardkonto festzulegen. Haben Sie nur ein einziges Outlook-Konto, was bei Verwendung eines Rechners für private und geschäftliche Zwecke unwahrscheinlich ist, haben Sie natürlich keine Wahl bei der Absenderadresse einer E-Mail. Wir gehen jedoch davon aus, dass viele Benutzer mindestens zwei Outlook-Konten in Outlook verwalten.

Einstellen des Absenderkontos

Outlook wählt also standardmäßig die E-Mail-Adresse des als Standard festgelegten Kontos beim Anlegen neuer E-Mails. Auf die gleiche Weise verfährt Outlook, wenn Sie wie im oben genannten Artikel eine neue E-Mail von einer .NET-Anwendung aus erstellen. Die einfachste Weise, das Absenderkonto zu ändern, erhalten Sie, wenn Sie die neu erstellte E-Mail mit der Display-

Methode anzeigen (siehe Bild 3). Das ist aber natürlich keine Option, wenn Sie die E-Mail direkt nach der Eingabe der Daten wie im Artikel **E-Mails mit Outlook verschicken** versenden wollen oder gar automatisiert E-Mails mit vorgefertigtem Inhalt an einen oder mehrere Empfänger gehen sollen. In diesem Fall benötigen Sie eine Möglichkeit, den Absender per Code einzustellen oder diesen zumindest gleich beim Zusammenstellen der Mail in der Anwendung festzulegen.

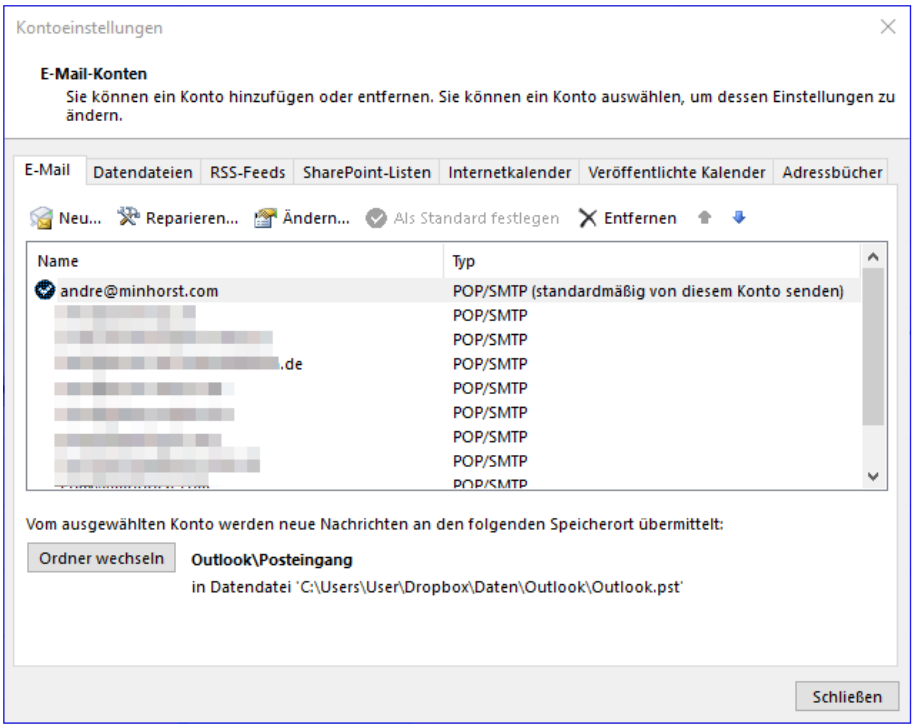


Bild 2: E-Mail-Konten unter Outlook

Absenderadresse festlegen

Wir werden uns zwei Varianten ansehen, um die Absenderadresse festzulegen. Bei der ersten geben wir die Absenderadresse einfach in ein Textfeld ein und versuchen dann beim Zusammenstellen der E-Mail, das betroffene Outlook-Konto zu ermitteln und als Absenderkonto einzustellen. Im zweiten Anlauf wollen wir dann vorab alle Outlook-Konten auslesen und die E-Mail-Adressen per Kombinationsfeld zur Auswahl bereitstellen.

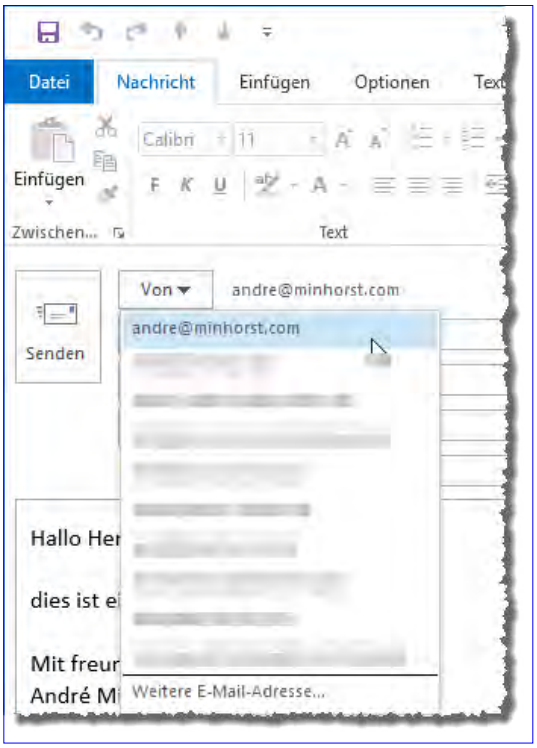


Bild 3: Auswahl des Absenders in der E-Mail

Wir bauen die Beispiellösung auf der Anwendung auf, die wir im Artikel **E-Mails mit Outlook verschicken** beschrieben haben. Dazu fügen wir zunächst ein Textfeld namens **txtAbsender** zum Fenster **MainWindow.xaml** hinzu:

```
<TextBox x:Name="txtAbsender" Grid.Column="1"></TextBox>
```

Damit dieses gleich beim Öffnen des Fensters gefüllt wird, fügen wir eine Zeile zur Methode **Window_Loaded** hinzu:

```
Private Sub Window_Loaded(sender As Object, e As RoutedEventArgs)
    txtAbsender.Text = "info@amvshop.de"
    ...
End Sub
```

E-Mails ohne Outlook versenden

Im Artikel »E-Mails mit Outlook verschicken« haben wir gezeigt, wie Sie von einer .NET-Anwendung aus die Daten für eine E-Mail zusammenstellen und diese dann per Outlook verschicken können. Was aber, wenn der Benutzer kein Outlook auf dem Rechner hat? Für diesen Fall bietet .NET eigene Bibliotheken. Damit können Sie E-Mails ohne Zugriff auf Outlook verschicken. Dieser Artikel zeigt, wie dies gelingt.

Wir wollen in diesem Artikel das gleiche Fenster verwenden, mit dem wir auch in der Beispiellösung zum Artikel **E-Mails mit Outlook verschicken** gearbeitet haben (siehe Bild 1). Einziger Unterschied: Die Schaltfläche **btnAnzeigen** fehlt, denn wir arbeiten ja nicht mit Outlook und können die Mail vor dem Versenden auch nicht mehr separat anzeigen. Dafür haben wir eine Schaltfläche namens Konfiguration hinzugefügt. Diese soll einen weiteren Dialog öffnen, der die Konfigurationsdaten für den Mailserver speichert.

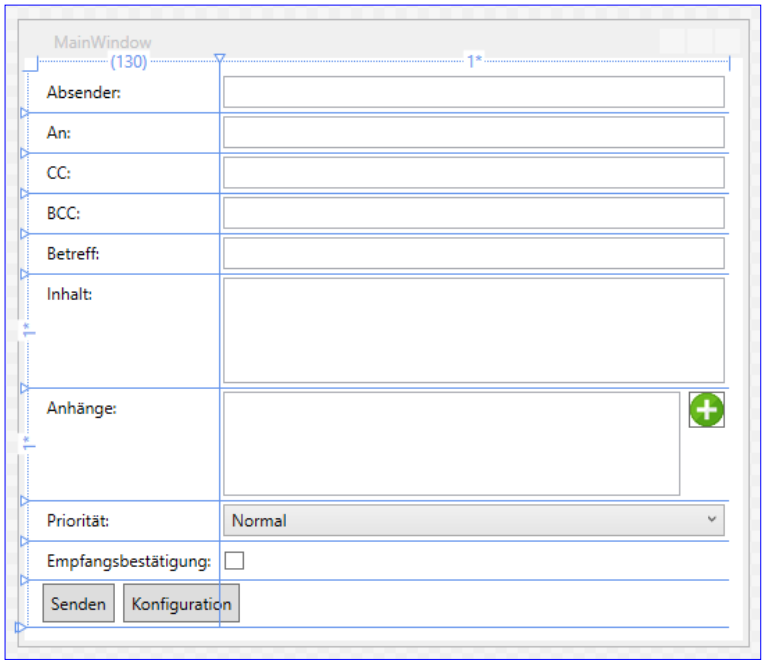


Bild 1: Fenster zur Eingabe der Mail-Daten

Für die Konfigurationsdaten legen wir einige Einträge im Bereich **Einstellungen** des aktuellen Projekts an, und zwar die folgenden (siehe Bild 2):

- **Host**: Name/IP des SMTP-Servers
- **Port**: Port des SMTP-Servers
- **Username**: Benutzername für die Anmeldung am SMTP-Server
- **Password**: Kennwort für die Anmeldung am SMTP-Server

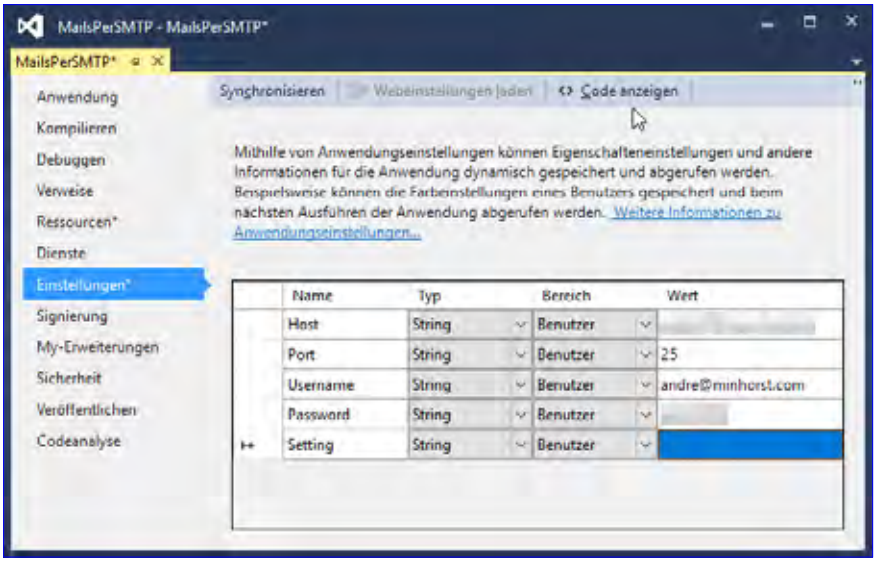


Bild 2: Anlegen der Konfigurationsdaten

Wenn Sie die Eigenschaften im Bereich **Einstellungen** angelegt haben und diese speichern, erstellt Visual Studio eine Klasse namens **MySettings**. Diese können Sie dann initialisieren und damit direkt die angelegten Eigenschaften ansprechen, also etwa **objSetting.User-**

name – mehr dazu weiter unten. Die Ereignismethode **Window_Load** füllt die Felder **txtAbsender**, **txtAn**, **txtBetreff** und **txtInhalt** mit Beispieldaten. Dies war beim Testen des Beispiels hilfreich, Sie werden diese Felder jedoch auf andere Weise füllen wollen – vorausgesetzt, Sie wollen überhaupt Mails über ein Fenster versenden und nicht direkt etwa mit den Daten aus einer Datenbank.

Erstellen und Senden der E-Mail

Die durch die Schaltfläche **btnSenden** ausgelöste Ereignismethode stellt die E-Mail zusammen und verschickt diese dann. Da wir hier nicht Outlook als Mail-Client verwenden, sondern die Klasse **MailMessage**, benötigen wir etwas mehr Code für die Durchführung der Aufgabe. Das liegt vorrangig daran, dass wir die Daten des SMTP-Servers sowie die Benutzerdaten erst noch angeben müssen, die ja bei Benutzung von Outlook als Mail-Client bereits im jeweiligen Outlook-Konto gespeichert sind. Um die nachfolgend verwendeten Klassen nutzen zu können, benötigen wir noch Verweise auf die folgenden Namespaces:

```
Imports System.Net.Mail
Imports System.Net
```

Die Methode deklariert zunächst einige Variablen und initialisiert dann ein neues Objekt auf Basis der Klasse **MailMessage**:

```
Private Sub btnSenden_Click(sender As Object, e As RoutedEventArgs)
    Dim objMail As MailMessage
    Dim objMailAddress As MailAddress
    Dim objAttachment As Attachment
    Dim objSMTPClient As SmtpClient
    Dim objNetworkCredentials As NetworkCredential
    Dim objSettings As MySettings
    objMail = New MailMessage
```

Dann beginnt die Methode, die Eigenschaften des neuen **MailMessage**-Objekts zu füllen. Als Absender geben wir für die Eigenschaft **From** ein Objekt des Typs **MailAddress** an, dem wir beim Initialisieren den Inhalt des Textfeldes **txtAn** übergeben:

```
    With objMail
        objMailAddress = New MailAddress(txtAn.Text)
        .From = objMailAddress
```

Die Empfängeradresse weisen wir mit der **Add**-Methode der Eigenschaft **To** zu. Betreff und Inhalt landen wie auch bei Outlook in den Eigenschaften **Subject** und **Body**:

```
        .To.Add(txtAn.Text)
        .Subject = txtBetreff.Text
        .Body = txtInhalt.Text
```

Die Anlagen werden auch etwas anders zur E-Mail hinzugefügt wir unter Verwendung von Outlook. Wir können der **Attachments**-Auflistung hier nicht einfach den Namen der hinzuzufügenden Datei zuweisen, sondern müssen erst ein Objekt des Typs

Mit Ressourcen arbeiten

In einer .NET-Anwendung können Sie auch ohne Verwendung einer Datenbank Dateien und andere Daten ablegen. Dazu stellt Microsoft die sogenannten Ressourcen (englisch Resources) zur Verfügung. Dabei handelt es sich um die Möglichkeit, Dateien, Texte oder andere Daten entweder über die Benutzeroberfläche von Visual Studio zur Anwendung hinzuzufügen und sie per Code auszulesen und gegebenenfalls auch wieder ins Dateisystem zu kopieren. Im konkreten Fall wollen wir eine Word-Dokumentvorlage in der Anwendung vorhalten, um diese bei Bedarf ins Dateisystem zu kopieren und als Vorlage zu nutzen.

Beispiel Word-Dokumentvorlage

Anlass zu diesem Artikel ist ein weiterer Artikel namens **Briefe mit Word erstellen**, in dem wir auf Basis einer Word-Dokumentvorlage ein neues Word-Dokument erstellen wollen.

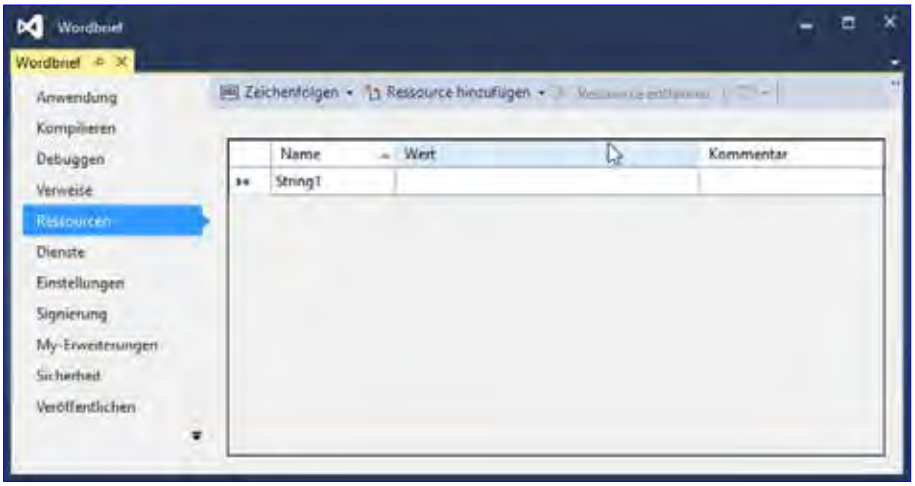


Bild 1: Der Dialog zum Verwalten der Ressourcen über Visual Studio

Für den Fall, dass der Benutzer die Anwendung frisch auf dem Rechner installiert hat, soll die Dokumentvorlage vor dem erstmaligen Erstellen von der Anwendung ins Dateisystem kopiert werden, damit der Code zum Erstellen des Word-Dokuments auf Basis dieser Vorlage auf die Vorlage zugreifen kann.

Ressourcen in Visual Studio verwalten

Der einfachste Weg, die Ressourcen zu verwalten, findet sich in der Benutzeroberfläche von Visual Studio. Wenn Sie ein Projekt angelegt haben, klicken Sie doppelt auf den Eintrag **My Project** im Projektmappen-Explorer, um die Eigenschaften der Anwendung anzuzeigen.

Diesen Bereich öffnen Sie auch mit dem Menübefehl **Projektl Eigenschaften**. Hier wechseln Sie dann durch einen Klick auf den Eintrag **Ressourcen** in der linken Leiste zum gleichnamigen Bereich. In einem frischen Projekt ist die Liste der Ressourcen noch leer (siehe Bild 1).

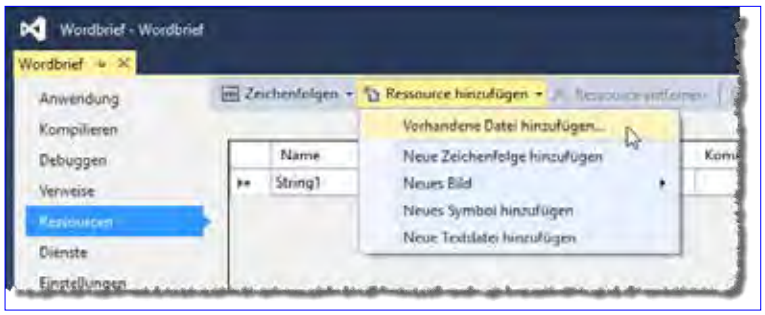


Bild 2: Hinzufügen einer Datei als Ressource

Um eine Datei als Ressource hinzuzufügen, klicken Sie auf den **Nach unten**-Pfeil der Schaltfläche **Ressource hinzufügen** im oberen Bereich und wählen im nun erscheinenden Menü den Eintrag **Vorhandene Datei hinzufügen...** aus (siehe Bild 2).

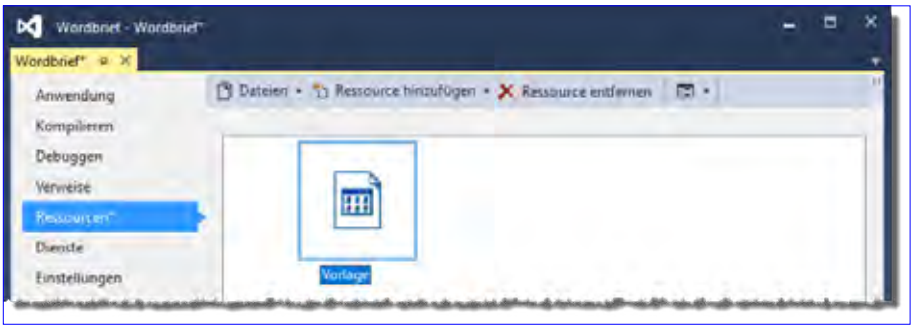


Bild 3: Anzeige der neuen Ressource in den Projekteigenschaften

Im folgenden Dialog **Vorhandene Datei zu Ressourcen hinzufügen** wählen Sie nun die hinzuzufügende Datei aus, in unserem Fall eine Datei namens **Vorlage.dotx**.

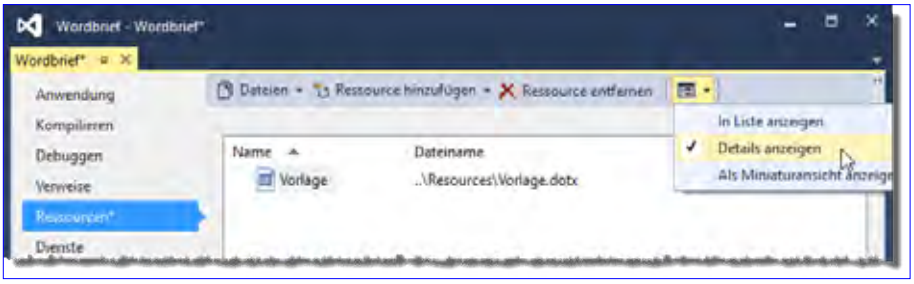


Bild 4: Detaillierte Liste der Ressourcen

Nach dem Hinzufügen zeigt der **Ressourcen**-Bereich die neue Ressource wie in Bild 3 an.

Diese Ansicht ist nicht besonders aussagekräftig, deshalb wechseln wir wie in Bild 4 über die Auswahl der Ansichten zur Ansicht **Details anzeigen**. Damit befindet sich die Ressource bereits in der Anwendung. Sie müssen nun noch die Änderungen speichern, am einfachsten mit der Tastenkombination **Strg + S**.

Ressource per Projektmappen-Explorer hinzufügen

Nach dem Hinzufügen der ersten Ressource zeigt der Projektmappen-Explorer einen Ordner für die Ressourcen an (siehe Bild 5). Dieser bietet auch ein Kontextmenü, über dessen Eintrag **Hinzufügen|Neues Element...** Sie den Dialog **Neues Element hinzufügen** öffnen. Hier können Sie allerdings nicht beliebige Dateien hinzufügen, sodass der Weg über den Ressourcen-Bereich in den Projekteigenschaften die einfachere Variante ist.

Zur Laufzeit mit Ressourcen arbeiten

Nun ist das primäre Ziel, diese Ressource zur Laufzeit aus der Anwendung ins Dateisystem zu kopieren. Dies wollen wir als Nächstes realisieren. Damit erschöpfen sich die Möglichkeiten auch: Die als Ressource

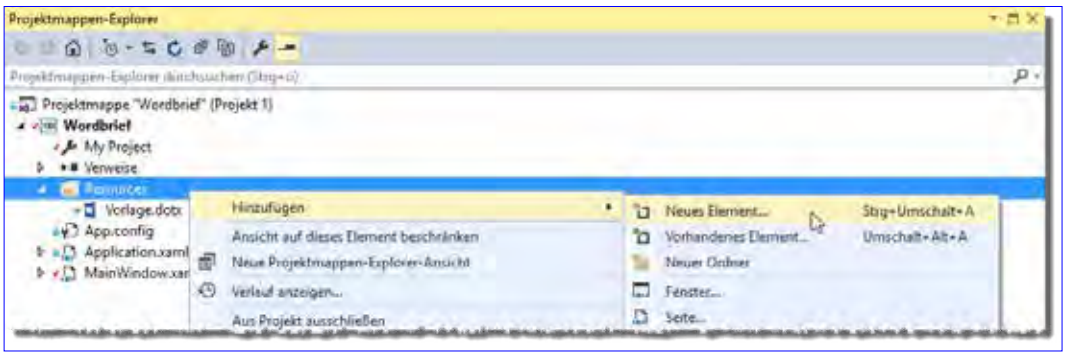


Bild 5: Ressource über den Projektmappen-Explorer hinzufügen

hinzugefügten Dateien werden nämlich beim Kompilieren in die **.exe**-Datei integriert. Sie können also nicht etwa zur Laufzeit weitere Elemente zu den Ressourcen einer Anwendung hinzufügen.

Schaltfläche zum Exportieren der Ressource

Für unsere erste Aufgabe fügen wir dem Fenster **MainWindow.xaml** eine Schaltfläche namens **btnRessourceExportieren** hinzu und legen für diese eine Ereignismethode an, die durch das Ereignis **Click** ausgelöst wird. Die Definition der Schaltfläche sieht so aus:

```
<Button x:Name="btnRessourceExportieren"
Click="btnRessourceExportieren_Click">
Ressource exportieren</Button>
```

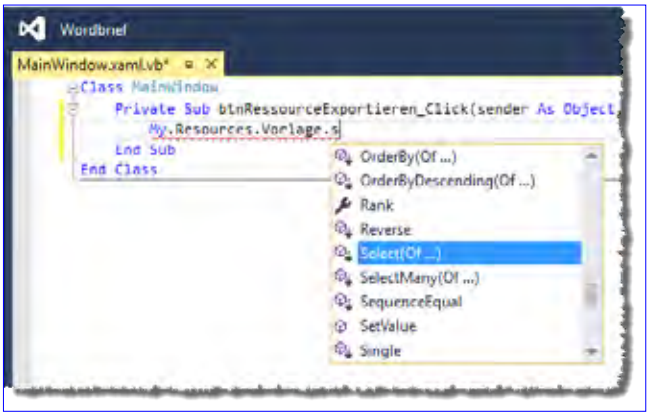


Bild 6: Zugriff auf die Ressource per Code

Wenn Sie nun zum Code-Editor für die Klasse **MainWindow.xaml.vb** wechseln, können Sie über **My.Resources** auf alle in den Ressourcen gespeicherten Elemente zugreifen, also auch auf unser Element mit der Bezeichnung **Vorlage**. Dieses bietet wiederum eine Reihe von Eigenschaften und Methoden an, unter denen sich aber leider nicht direkt eine Eigenschaft namens **SaveAs** findet (siehe Bild 6).

Wie bekommen wir die als Ressource gespeicherte Datei **Vorlage.dotx** nun dennoch in unser Dateisystem? Dazu brauchen wir die Methode lediglich wie folgt zu erweitern:

```
Private Sub btnRessourceExportieren_Click(sender As Object, e As RoutedEventArgs)
    Dim objVorlage As Byte()
    objVorlage = My.Resources.Vorlage
    My.Computer.FileSystem.WriteAllBytes(AppDomain.CurrentDomain.BaseDirectory + "\Vorlage.dotx", objVorlage, False)
End Sub
```

Die erste Zeile deklariert ein Byte-Array, die zweite weist diesem den Inhalt der Ressource zu, die wir mit **My.Resources.Vorlage** referenzieren. Die dritte speichert die Ressource unter dem Dateinamen **Vorlage.dotx** im aktuellen Verzeichnis der **.exe**-Anwendung.

Zielverzeichnis definieren

Nun wollen wir dem Benutzer noch die Möglichkeit geben, die Ressource in einem von ihm ausgewählten Verzeichnis zu speichern. Dazu legen wir eine zweite Schaltfläche namens **btnRessourceExportierenAls** an, für die wir die folgende Methode hinterlegen:

```
Private Sub btnRessourceExportierenAls_Click(sender As Object, e As RoutedEventArgs)
    Dim objVorlage As Byte()
    Dim objSaveFileDialog As SaveFileDialog
```

Brief mit Word erstellen

Heute war es wieder soweit: Ich musste ein Anschreiben erstellen. Heutzutage geht zwar vieles per E-Mail, aber hier und da wird doch noch nach einem korrekten Anschreiben mit Briefkopf, Adresse, Ort und Datum, Betreff und dem Text selbst verlangt. Da dies so selten passiert, finde ich meist meine Wordvorlage für diesen Zweck nicht: Seit dem letzten Anschreiben kann es nämlich gut sein, dass ich einen neuen oder neu aufgesetzten Rechner habe und ich vergessen habe, meine Vorlagen vom alten Rechner auf den neuen zu übertragen. Wie schön wäre es doch, wenn man eine kleine .NET-Anwendung hätte, die einem ein paar Textfelder für die wichtigsten Daten bereitstellt und nach der Eingabe per Mausklick das gewünschte Dokument im .docx-Format anlegt! Wie das gelingt, zeigt Ihnen der vorliegende Artikel.

Wenn Sie ein Dokument wie oben beschrieben erstellen möchten, erledigen Sie das üblicherweise auf Basis einer Dokumentvorlage, die bereits einige wichtige Informationen enthält – zum Beispiel, wo die einzelnen Elemente wie Briefkopf, Anschrift, Ort und Datum, Betreff und Inhalt angelegt werden. Genau diese Dokumentvorlage benötigen wir auch für die Beispiellösung zu diesem Beitrag. Dumm nur, dass sich genau das Mitnehmen solcher Dokumente beim Aufsetzen eines neuen Rechners bei mir als Problem herausstellte und ich dann, statt die Vorlage auf dem alten Rechner zu suchen, lieber schnell von Hand ein neues Anschreiben zusammengesetzt habe. Also sollte die Lösung auch die Möglichkeit bieten, zumindest eine Dokumentvorlage zu speichern und diese dann bei Bedarf bereitzustellen. Wie dies gelingt, zeigen wir ausführlich im Artikel **Mit Resourcen arbeiten**.

Die grobe Vorgehensweise sieht so aus: Wir haben eine Anwendung, welche eine Word-Dokumentvorlage als Ressource enthält. Die Benutzeroberfläche stellt Textfelder bereits, mit denen Sie die im Anschreiben auszugebenden Informationen eintippen können. Nach der Eingabe können Sie per Mausklick einen Vorgang starten, bei dem die Anwendung die Dokumentvorlage im Dateisystem speichert (sofern diese noch nicht vorhanden ist) und ein neues Dokument auf Basis dieser Vorlage erstellt und mit den in der Anwendung eingegebenen Daten füllt. Das Dokument soll anschließend geöffnet werden, damit der Benutzer es beispielsweise ausdrucken kann. Alternativ wollen wir noch eine Schaltfläche anlegen, mit welcher der Benutzer direkt ein PDF-Dokument auf Basis des erstellten Word-Dokuments erzeugen kann.

Dokumentvorlage vorbereiten

Der erste Schritt ist das Vorbereiten einer Dokumentvorlage. Das heißt, dass wir in Microsoft Word ein neues Dokument anlegen, die gewünschten Bereiche inklusive Platzhalter definieren und beispielsweise den Briefkopf hinzufügen und das Dokument dann unter der Dateiendung **.dotx** speichern. Letzteres ist besonders wichtig, damit wir neue Dokumente auf Basis dieser Vorlage anlegen können. In welcher Form wollen wir die Platzhalter anlegen? Am einfachsten ist es, einfach entsprechende Texte in eckigen Klammern zu den jeweiligen Elementen hinzuzufügen, also beispielsweise **[Betreffzeile]**.

Vorlage nach DIN 5008

Wenn wir schon eine neue Vorlage anlegen, wollen wir diese auch halbwegs DIN-gerecht erstellen. Deshalb schauen wir uns dabei die aktuellen Vorgaben entsprechend DIN 5008 an, die etwa unter **www.din-5008-richtlinien.de** schön zusammenge-

fasst sind. Wir beginnen mit dem Anschriftenfeld, für das die DIN-Norm angibt, dass es 40 x 85 mm groß ist und einen Abstand von 45 mm zum oberen und 20 mm zum linken Rand aufweist (es gibt noch eine Variante mit 27 mm zum oberen Rand, aber wir wollen die Version für Briefumschläge mit Adressfenster nutzen).

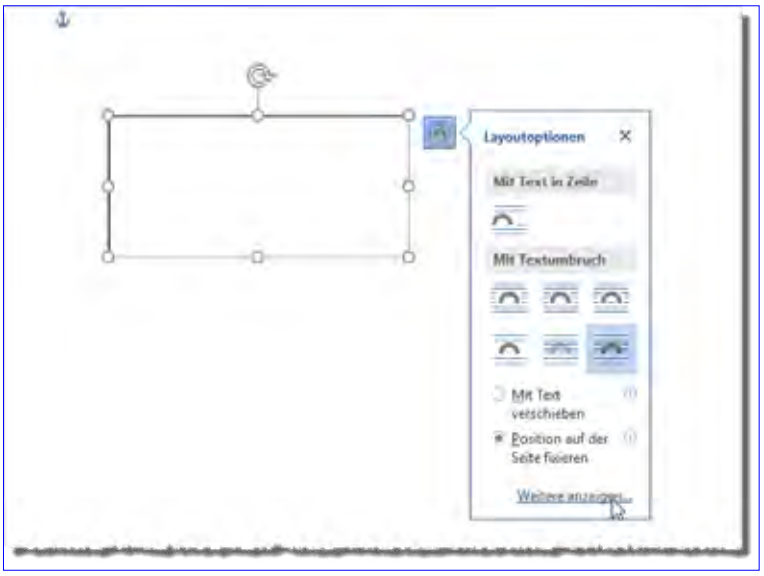


Bild 1: Anzeigen der Textfeld-Eigenschaften

Anschriftenfeld

Wir öffnen also Word und legen ein neues, leeres Dokument an. Für das Anschriftenfeld fügen Sie dann ein Textfeld hinzu. Dies erledigen wir mit dem Ribbon-Eintrag **Einfügen|Text|Textfeld|Textfeld** erstellen. Ziehen Sie nach Betätigung dieses Befehls einen Rahmen auf, der etwa den gewünschten Abmessungen entspricht. Klicken Sie dann auf die Schaltfläche, die rechts oberhalb vom neuen Textfeld erscheint, und wählen Sie unten den Befehl **Weitere anzeigen** aus (siehe Bild 1).

Bevor wir auf den Dialog **Layout** schauen, noch die Information, dass Word mitunter etwas verwirrend ist (Version 2016). Word zeigt nämlich standardmäßig den verfügbaren Bereich ohne Seitenränder an – zumindest ohne die oberen und unteren Seitenränder. Wenn man das weiß, kann man die Einstellungen im Layout-Dialog wie in Bild 2 ohne Rücksicht darauf vornehmen, dass die Position des Textfeldes in der Ansicht Drucklayout nicht so aussieht, wie sie später auf dem Papier erscheinen wird – also mit 2 cm Abstand zur Seite horizontal und mit 4,5 cm Abstand zur Seite vertikal.

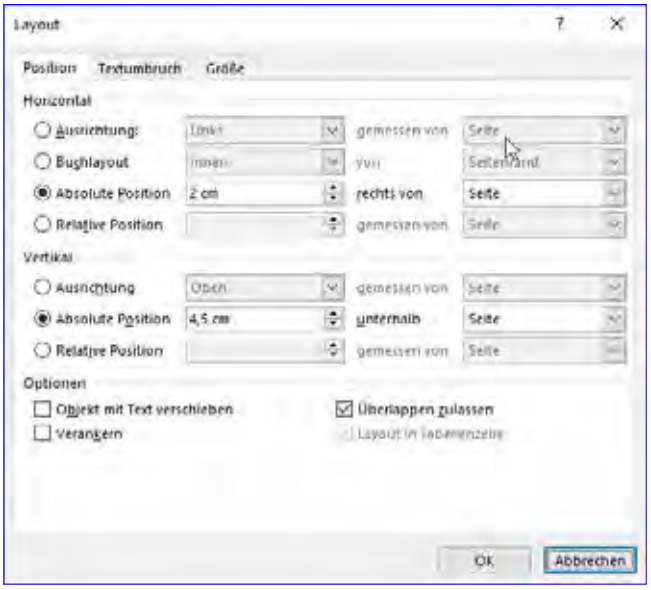


Bild 2: Position des Textfelds

Die Größe des Textfeldes stellen wir analog im Bereich Größe des gleichen Dialogs ein – und zwar wie von DIN 5008 vorgegeben auf 4 cm x 8,5 cm (siehe Bild 3). Die Größe können Sie allerdings auch über die beiden Steuerelemente in der Gruppe **Format|Größe** des Ribbons einstellen.

Bild 3: Größe des Textfelds

Schließlich weist das Textfeld noch einen Rahmen auf, den wir transparent machen wollen. Dazu wählen Sie beim markiertem Textfeld den Ribbon-Eintrag **Format|Formenarten|Formkontur|Keine Kontur** aus.

Bild 4: Einfügen einer Textmarke über das Ribbon

Ganz oben im Anschriftenfeld fügen wir nun eine Zeile mit der Rücksendeadresse ein, also der Adresse, an die der Brief zurückgesendet werden soll, wenn dieser nicht zugestellt werden konnte. Für diese Zeile legen Sie eine recht kleine Schriftgröße fest, zum Beispiel **8**. Darunter soll die eigentliche Anschrift landen – wie, erfahren Sie in den nächsten Abschnitten.

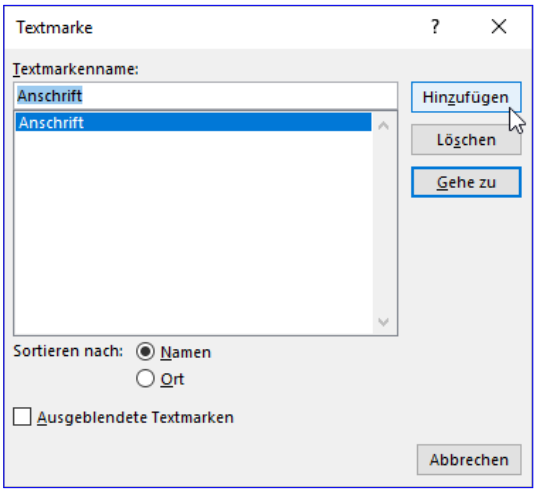


Bild 5: Einfügen einer Textmarke

Textmarke einfügen

Später wollen wir den Textfeldern Texte hinzufügen. Dazu hinterlegen wir in den Textfeldern jeweils eine Textmarke, die wir dann später per Code leicht auffinden und durch die gewünschten Texte ersetzen können. Um die Textmarke hinzuzufügen, platzieren Sie die Einfügemarke im Textfeld für die Anschrift gleich unter der Rücksendeadresse und stellen die Schriftart auf die gewünschte Größe ein, beispielsweise auf **10**. Danach fügen wir die Textmarke ein. Dies erledigen wir mit dem Ribbon-Befehl **Einfügen|Links|Textmarke** (siehe Bild 4).

Dies öffnet den Dialog aus Bild 5, mit dem wir den Namen der neuen Textmarke eingeben und diese dann mit der Schaltfläche **Hinzufügen** an der Position der Einfügemarke einfügen. Im Dokument ist die Textmarke nun zunächst nicht zu erkennen. Das Vorhandensein dokumentiert lediglich der neue Eintrag im Dialog **Textmarke**.

Zur Sicherheit wollen wir die Textmarke jedoch noch sichtbar machen. Dazu öffnen Sie über das Ribbon den Dialog **Word-Optionen** und klicken dort links auf den Eintrag **Erweitert**. Im rechten Bereich finden Sie dann unter Dokumentinhalt anzeigen die Option **Textmarken anzeigen**, die Sie nun aktivieren (siehe Bild 6).

Wenn Sie den Dialog nun mit einem Klick auf die Schaltfläche

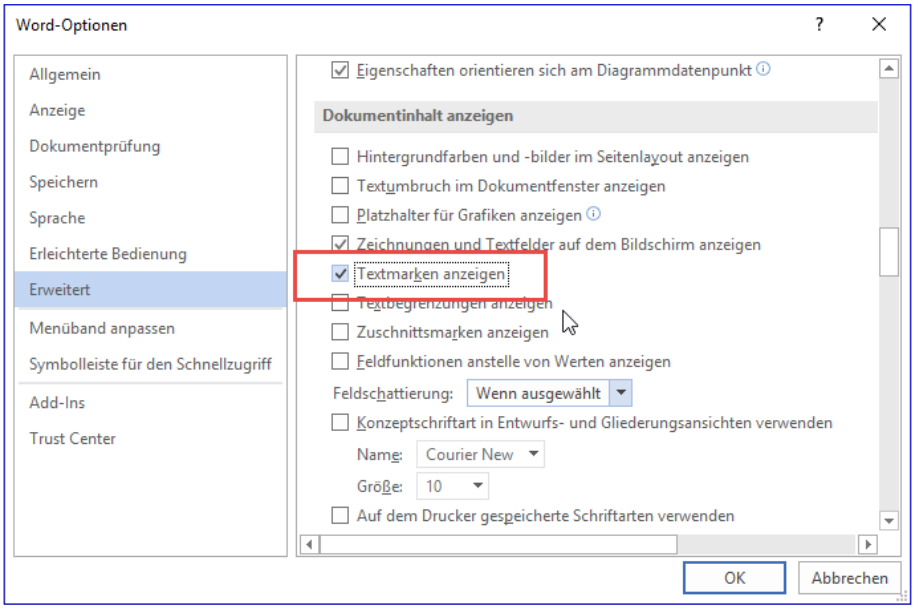


Bild 6: Textmarken einblenden

OK schließen, finden Sie im Textfeld für die Anschrift die Textmarke wie in Bild 7 vor.

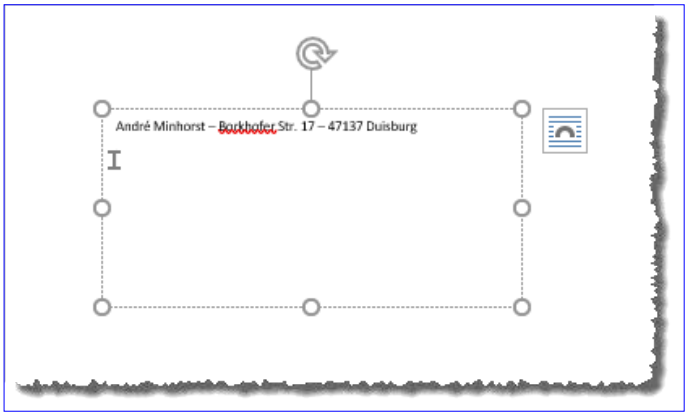


Bild 7: Markierung für eine Textmarke

Wenn Sie später alle Textfelder sichtbar machen wollen, können Sie einfach in einen leeren Bereich des Dokuments klicken und alle Elemente mit der Tastenkombination **Strg + A** markieren.

Das Textfeld können Sie nun gleich kopieren und als Basis für das Feld für die Betreffzeile nutzen – so brauchen Sie den Rahmen nicht wieder wie bei einem neuen Textfeld auf transparent einzustellen. Die Betreffzeile soll zwei Zeilen unterhalb des Anschriftenfeldes liegen, was wir als ca. 10 mm interpretieren. Also stellen wir den Abstand vom oberen Rand auf 10,5 cm ein, den Abstand vom linken Rand auf 2 cm und die Größe auf 1 x 12 cm. Löschen Sie den mitkopierten Text, stellen Sie die Schriftgröße auf 10 ein und fügen Sie auch diesem Textfeld einen Platzhalter hinzu – diesmal mit der Bezeichnung **Betreff**.

Rechts oben wollen wir den Briefkopf unterbringen. Dafür legen wir ein Textfeld mit einem Abstand von 2 cm vom rechten Rand und 2,5 cm vom oberen Rand und mit der Größe 5 x 5 cm an. Dazu müssen wir ein wenig rechnen, denn wir können nur Abstände von oben und links eintragen. Ein DIN A4-Blatt hat eine Breite von 19,7 cm, also müssen wir 2 cm für den Abstand von rechts und 5 cm für die Breite abziehen und landen bei einem Abstand von 12,7 cm vom linken Rand. Den Briefkopf füllen Sie mit den Absenderdaten aus und mit weiteren Daten, die Sie angeben möchten – Telefon, E-Mail, Webseite, Bankverbindung, Steuernummern et cetera.

Das Datum des Anschreibens fügen wir in einem Textfeld ein, das die Position 12,7 cm vom linken Rand und 9 cm vom oberen Rand einnimmt und eine Größe von 5 x 1 cm aufweist. Das Textfeld füllen wir mit einer weiteren Textmarke, diesmal mit der Bezeichnung **Datum** in Schriftgröße 10.

Das letzte Textfeld soll den Inhalt des Anschreibens aufnehmen. Es hat einen Abstand von 2 cm zum linken Rand und 13 cm zum oberen Rand und eine Größe von 15,7 x 12 cm. Hier fügen wir die Textmarke mit der Bezeichnung **Inhalt** ein.

An dieser Stelle der Hinweis, dass wir hier nur in Textfeldern schreiben und daher unabhängig von den Einstellungen der Seitenränder des Dokuments sind. Damit erhalten wir allerdings die Einschränkung, dass wir auch nur den Platz des dafür vorgesehenen Textfeldes für den Inhalt des Anschreibens haben. Für mehrseitige Dokumente müssten wir allerdings auch noch etwas mehr Aufwand betreiben – das soll jedoch nicht Thema dieses Artikels sein. Das Dokument sieht schließlich wie in Bild 8 aus. Schließlich speichern Sie die Word-Datei unter dem Namen **Vorlage.dotx**, wozu Sie den entsprechenden Dateityp beim Speichern auswählen müssen.

Vorlage wieder öffnen

Wenn Sie die Dokumentvorlage nach dem Schließen nochmals öffnen wollen, um diese beispielsweise nochmals anzupassen, können Sie dies nicht einfach per Doppelklick auf die Datei **Vorlage.dotx** erledigen. Dies öffnet nämlich nicht die Dokument-

vorlge, sondern erzeugt ein neues Dokument auf Basis dieser Vorlage.

Dieses sieht dann zwar genauso aus wie die Vorlage selbst, aber wenn Sie diese speichern, stellen Sie fest, dass es sich hier um ein einfaches Word-Dokument handelt.

Wenn Sie die Vorlagendatei öffnen wollen, starten Sie zunächst Word und öffnen die Vorlagendatei dann über den **Datei öffnen**-Dialog von Word.

Steuerelemente zum Eingeben der Briefdaten

Damit kommen wir zurück zum Visual Studio-Projekt. Diesem fügen wir nun zunächst die fertige Word-Vorlage als Ressource hinzu. Dazu öffnen Sie die Projekteigenschaften, wählen den Eintrag **Ressourcen** und legen die Word-Dokumentvorlage als neue Ressource an. Details dazu finden Sie im Artikel **Mit Ressourcen arbeiten**.

Wir machen es auf die Schnelle so, dass wir einfach die Datei **Vorlage.dotx** aus dem Windows Explorer in die Liste der Ressourcen hineinziehen (siehe Bild 9).

Sie können natürlich auch noch weitere Vorlagen anlegen, die beispielsweise für andere Familienmitglieder genutzt werden oder für geschäftliche Zwecke.

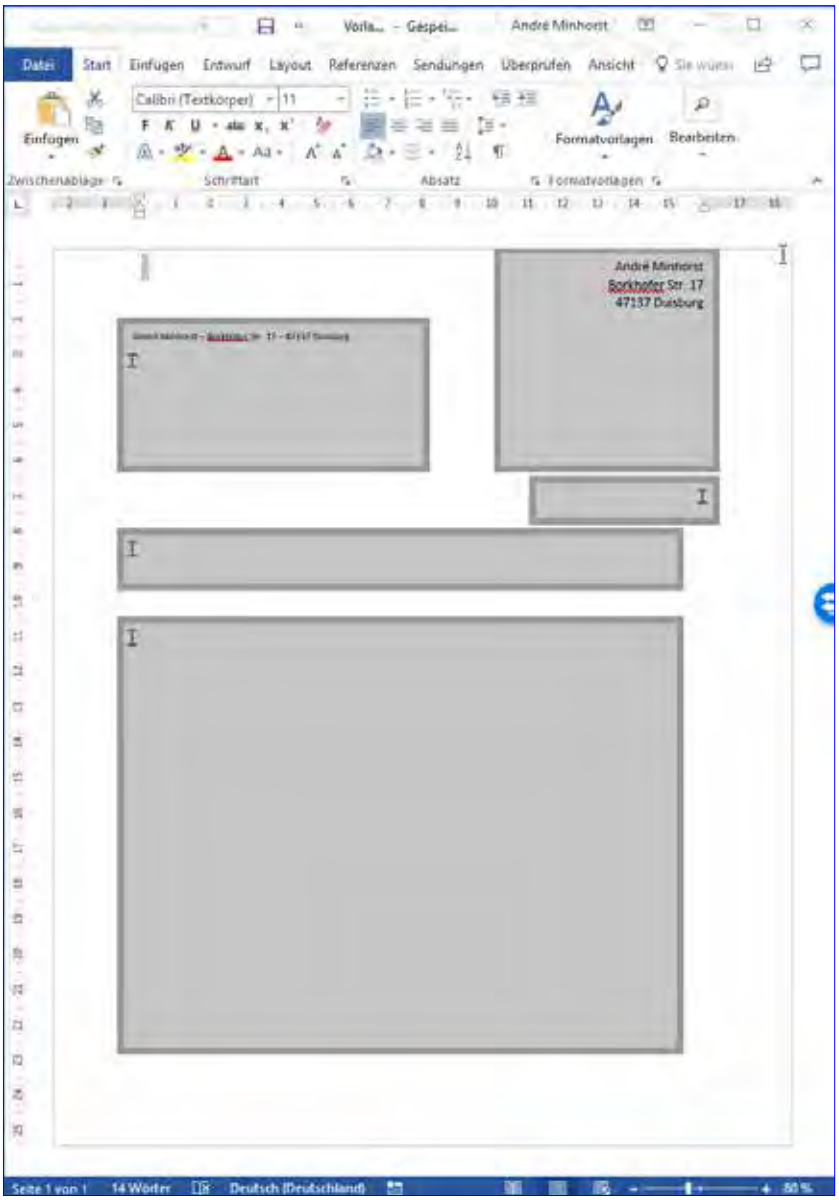


Bild 8: Anordnung der Textfelder

Anzeige und Auswahl der Vorlagen

Ganz oben im Fenster wollen wir ein Kombinationsfeld zur Auswahl der Vorlage einfügen. Das Label und das Kombinationsfeld definieren wir wie folgt:

```
<ComboBox x:Name="cboVorlagen" Grid.RowSpan="2" Grid.Column="1"/>
<Button x:Name="btnGewaehlteRessourcenExportieren" Click="btnGewaehlteRessourcenExportieren_Click"
    Grid.Row="8">Gewählte Ressource exportieren</Button>
```

E-Mails mit Outlook verschicken

Wer Anwendungen mit Visual Studio programmiert, die Daten wie etwa die von Kunden verwaltet, kommt früher oder später nicht um eine Funktion zum Versenden von E-Mails herum. Da gibt es nun zwei Möglichkeiten: Sie verwenden eine eigene SMTP-Klasse, um die E-Mails zu versenden. Das klappt mit .NET-Projekten viel einfacher als etwa unter Access, weil es hier schon fertige Klassen für diesen Anwendungszweck gibt. Allerdings werden die gesendeten Mails dann nicht in Outlook im Ordner »Gesendete Elemente« gespeichert. Deshalb schauen wir uns in diesem Artikel den Versand von Mails per Outlook an. In einem anderen Artikel gehen wir dann auf die Verwendung einer SMTP-Klasse ein.

Beispielanwendung

Wir wollen als Beispiel für diesen Artikel eine kleine WPF-Anwendung programmieren, mit der Sie die Daten des E-Mail-Empfängers sowie Betreff, Inhalt und weitere Informationen eingeben können. Eine Schaltfläche soll dann die eingegebenen Daten an eine Outlook-Instanz schicken und die E-Mail versenden.

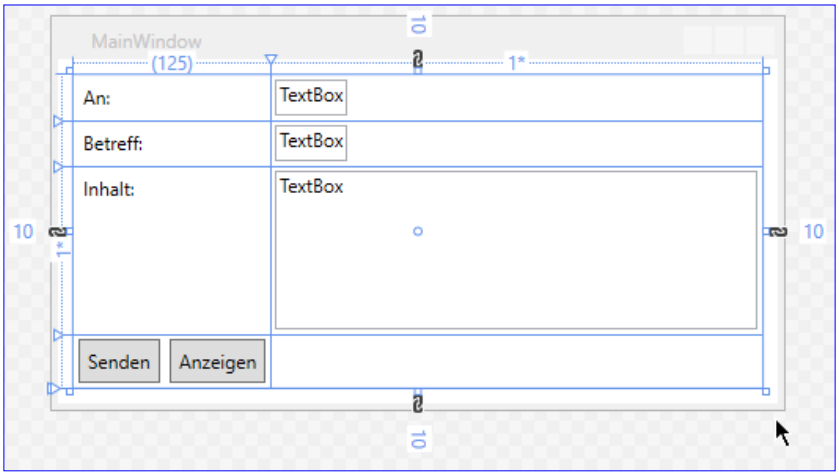


Bild 1: Einfaches Formular zum Versenden von E-Mails

Also legen Sie zunächst eine WPF-Anwendung in der von Ihnen bevorzugten Sprache an (also etwa VB oder C#). Das Fenster **MainWindow.xaml** ist dann auch gleich das Element, das wir mit den benötigten Steuerelementen versehen wollen. Dieses gestalten wir wie in Bild 1.

Der dazu notwendige XAML-Code sieht gekürzt so aus (es gibt noch einige Steuerelement-spezifische Eigenschaftsdefinitionen, die wir nicht abbilden):

```
<Window x:Class="MainWindow" ... Title="MainWindow" Height="250" Width="466">
    <Grid HorizontalAlignment="Stretch" Margin="10" VerticalAlignment="Stretch" ClipToBounds="true" >
        <Grid.RowDefinitions>
            <RowDefinition Height="Auto"/>
            <RowDefinition Height="Auto"/>
            <RowDefinition/>
            <RowDefinition Height="Auto"/>
            <RowDefinition Height="Auto"/>
        </Grid.RowDefinitions>
```

```
        <Grid.ColumnDefinitions>
            <ColumnDefinition Width="Auto"/>
            <ColumnDefinition Width="*"/>
        </Grid.ColumnDefinitions>
        <Label x:Name="label1" Content="An:" />
        <Label x:Name="label2" Content="Betreff:" Grid.Row="1"/>
        <Label x:Name="label3" Content="Inhalt:" Grid.Row="2"/>
        <TextBox x:Name="txtAn" Grid.Column="1" HorizontalAlignment="Stretch"/>
        <TextBox x:Name="txtBetreff" Grid.Row="1" Grid.Column="1" HorizontalAlignment="Stretch"/>
        <TextBox x:Name="txtInhalt" Grid.Row="2" TextWrapping="Wrap" Grid.Column="1" VerticalAlignment="Stretch"
            HorizontalAlignment="Stretch" Height="Auto" AcceptsReturn="True"/>
        <StackPanel Orientation="Horizontal" Grid.Row="3">
            <Button x:Name="btnSenden">Senden</Button>
            <Button x:Name="btnAnzeigen">Anzeigen</Button>
        </StackPanel>
    </Grid>
</Window>
```

Dadurch, dass wir für die dritte Zeile den Wert der Eigenschaften **Height** und **Width** auf **Auto** eingestellt haben, wird diese Zeile gemeinsam mit dem Fenster vergrößert. Damit wir im Textfeld **txtInhalt** mehrzeilige Texte eingeben können, haben wir die Eigenschaft **TextWrapping** auf **Wrap** und **AcceptsReturn** auf **True** eingestellt.

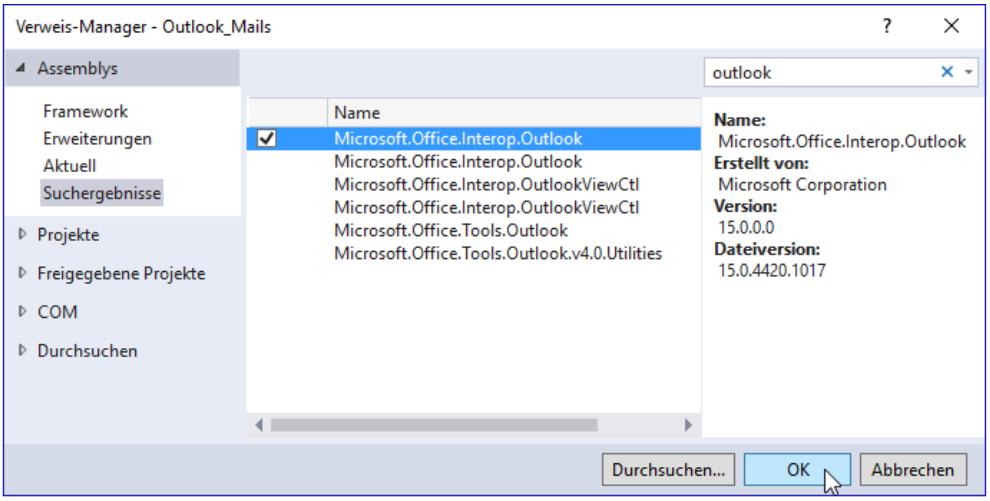


Bild 2: Erweitern der Verweise des Projekts um die Outlook-Bibliothek

Verweis auf Outlook hinzufügen

Damit wir Outlook von einer WPF-Anwendung aus programmieren können, fügen wir den entsprechenden Verweis auf die Outlook-Bibliothek **Microsoft.Office.Interop.Outlook** hinzu. Dazu öffnen Sie den **Verweise-Manager** über den Menüeintrag **Projekt|Verweis hinzufügen...** und suchen im nun erscheinenden Dialog nach dem Wort **Outlook**.

Selektieren Sie den Eintrag **Microsoft.Office.Outlook.Interop** (siehe Bild 2) und schließen Sie den Dialog mit **OK**. Sollten Sie, wie im obigen Dialog der Fall, mehrere gleichnamige Einträge vorfinden, handelt es sich um verschiedene Versionen. Wählen Sie dann die aktuellere Version aus.

Dann statten wir die Schaltfläche **cmdSenden** mit der Ereigniseigenschaft **Click** aus. Nach der Eingabe des Gleichheitszeichens können Sie mit der Tabulator-Taste automatisch die entsprechende Ereignismethode anlegen, indem Sie aus der dann erscheinenden Auswahlliste den Eintrag **<Neuer Ereignishandler>** auswählen:

```
<Button x:Name="btnSenden" Click="btnSenden_Click">Senden</Button>
```

In der Code behind-Klasse erscheint dann die folgende Methode:

```
private void btnSenden_Click(object sender, RoutedEventArgs e) {

}
```

In der Klasse fügen wir nun zunächst eine **Imports**-Anweisung hinzu, mit der wir den Outlook-Namespace verfügbar machen:

```
using Outlook = Microsoft.Office.Interop.Outlook;
```

Damit greifen wir über das Schlüsselwort **Outlook** auf Elemente dieses Namespaces zu.

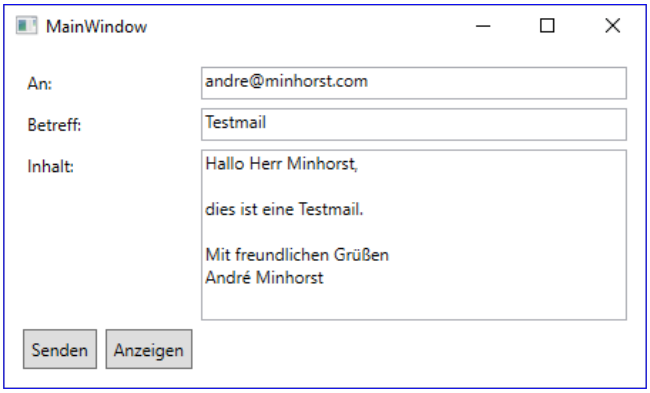


Bild 3: Beispielfenster zum Versenden von E-Mails

Textfelder mit Beispiel füllen

Bevor wir uns auf die Outlook-Programmierung stürzen, legen wir noch eine Ereignismethode für das Ereignis **Loaded** des **Window**-Elements an. Diese soll die drei Textfelder mit Beispieltexten füllen, damit das Fenster beim Öffnen gleich wie in Bild 3 aussieht:

```
private void Window_Loaded(object sender, RoutedEventArgs e) {
    txtAn.Text = "andre@minhorst.com";
    txtBetreff.Text = "Testmail";
    txtInhalt.Text = "Hallo Herr Minhorst," +  Environment.NewLine + Environment.NewLine + "dies ist eine Testmail."
        + Environment.NewLine + Environment.NewLine + "Mit freundlichen Grüßen" + Environment.NewLine + "André Minhorst";
}
```

Outlook-Mail erzeugen

Danach kümmern wir uns um den Code, der die Mail mit den angegebenen Daten versenden soll. Dazu füllen wir die bereits angelegte Ereignismethode **btnSenden_Click** wie folgt:

```
private void btnSenden_Click(object sender, RoutedEventArgs e) {
    Outlook.Application objOutlook;
    Outlook.MailItem objMailItem;
    objOutlook = new Outlook.Application();
```

```
    objMailItem = objOutlook.CreateItem(Outlook.OlItemType.olMailItem);
    objMailItem.To = txtAn.Text;
    objMailItem.Subject = txtBetreff.Text;
    objMailItem.Body = txtInhalt.Text;
    objMailItem.Send();
}
```

Wir erstellen also ein Objekt des Typs **Microsoft.Office.Interop.Outlook.Application**, den wir hier aufgrund unserer **Using**-Anweisung vereinfacht mit **Outlook.Application** referenzieren können, und speichern einen Verweis darauf in der Variablen **objOutlook**. Die zweite Variable namens **objMailItem** mit dem Datentyp **MailItem** nutzen wir, um ein mit der **CreateItem**-Methode von **objOutlook** erzeugtes **MailItem**-Objekt zu referenzieren. Dies dient dem Zweck, seine Eigenschaften **To**, **Subject** und **Body** mit den Inhalten der drei Textfelder **txtAn**, **txtBetreff** und **txtInhalt** zu füllen. Die letzte Anweisung heißt **Send** und sorgt dafür, dass die Mail direkt verschickt wird. Wenn Sie die Anwendung nun starten und die Mail mit einem Klick auf die Schaltfläche **cmdSenden** verschicken, können Sie diese anschließend im Ordner **Gesendete Objekte** von Outlook einsehen.

E-Mail erst anzeigen

Etwas transparenter ist das Ganze natürlich noch, wenn Sie die E-Mail betrachten können, bevor Sie diese durch einen Klick auf die dafür vorgesehene Schaltfläche absenden. Dazu legen wir für die Schaltfläche **cmdAnzeigen** ebenfalls die **Click**-Eigenschaft an:

```
<Button x:Name="btnAnzeigen" Click="btnAnzeigen_Click">Anzeigen</Button>
```

Den Code dieser Methode brauchen wir gegenüber der zuvor beschriebenen nur leicht zu variieren:

```
private void btnSenden_Click(object
sender, RoutedEventArgs e) {
    ...
    objMailItem.Send();
}
```

Hier rufen wir nämlich statt der **Send**-Methode die **Display**-Methode auf. Dies zeigt beim Ausprobieren die Mail aus Bild 4 an. Nach dem Betrachten schickt man die E-Mail dann mit einem Klick auf die **Senden**-Schaltfläche ab.

Bild 4: E-Mail vor dem Versenden ansehen

Anhänge hinzufügen

Nun wollen wir dem Benutzer ermöglichen, einen oder mehrere Dateien zur

Absender von Outlook-Mails einstellen

Im Artikel »E-Mails mit Outlook verschicken« haben wir gezeigt, wie Sie von einer .NET-Anwendung aus die Daten für eine E-Mail zusammenstellen und diese dann per Outlook verschicken können – unter anderem mit Empfänger, CC, BCC, Anlagen oder Priorität. Was noch fehlt, ist die Absenderadresse. Diese braucht man grundsätzlich nicht explizit einzustellen, denn Outlook verwendet dann die Adresse des Standardkontos von Outlook (vorausgesetzt, Sie haben mehrere Konten). Wenn Sie jedoch eine Mail mit einem anderen Konto als dem Standardkonto verschicken wollen, wird es kompliziert. Die Lösung für diese Aufgabe finden Sie im vorliegenden Artikel.

Verschiedene Outlook-Konten

Wenn Sie wie im **Artikel E-Mails mit Outlook verschicken** ein **MailItem**-Objekt erzeugen und seine Eigenschaften wie **To**, **CC**, **BCC**, **Subject** oder **Body** füllen, werden Sie vergeblich ein Feld etwa namens **From** suchen. Dieses Feld gibt es nämlich nicht – Sie können also nicht einfach irgendeine Absenderadresse zu Ihrer E-Mail hinzufügen.

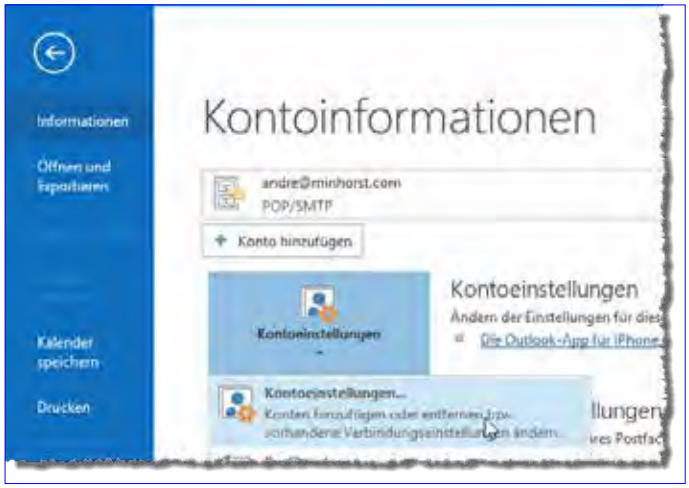


Bild 1: Anzeigen der Kontoeigenschaften

Woher weiß Outlook dann aber, welche Absenderadresse überhaupt zu verwenden ist? Ganz einfach: Es gibt in Outlook ein oder mehrere Konten, von denen eines als Standard festgelegt ist. Wann immer Sie direkt von Outlook aus eine neue E-Mail erstellen, wird die zum Standardkonto gehörende E-Mail-Adresse als Absenderadresse eingestellt. Wo finden Sie diese Konten und Einstellungen? Dazu öffnen Sie Outlook, klicken auf den Ribbon-Tab **Datei** und dann im Bereich **Informationen** auf die Schaltfläche **Kontoeinstellungen|Kontoeinstellungen...** (siehe Bild 1).

Hier finden Sie dann wie in Bild 2 die angelegten Outlook-Konten vor. Hier können Sie beispielsweise doppelt auf einen der Einträge klicken, um Eigenschaften wie den SMTP-Server oder die E-Mail-Adresse anzupassen oder ein anderes als das aktuelle Konto als Standardkonto festzulegen. Haben Sie nur ein einziges Outlook-Konto, was bei Verwendung eines Rechners für private und geschäftliche Zwecke unwahrscheinlich ist, haben Sie natürlich keine Wahl bei der Absenderadresse einer E-Mail. Wir gehen jedoch davon aus, dass viele Benutzer mindestens zwei Outlook-Konten in Outlook verwalten.

Einstellen des Absenderkontos

Outlook wählt also standardmäßig die E-Mail-Adresse des als Standard festgelegten Kontos beim Anlegen neuer E-Mails. Auf die gleiche Weise verfährt Outlook, wenn Sie wie im oben genannten Artikel eine neue E-Mail von einer .NET-Anwendung aus erstellen. Die einfachste Weise, das Absenderkonto zu ändern, erhalten Sie, wenn Sie die neu erstellte E-Mail mit der Display-

Methode anzeigen (siehe Bild 3). Das ist aber natürlich keine Option, wenn Sie die E-Mail direkt nach der Eingabe der Daten wie im Artikel **E-Mails mit Outlook verschicken** versenden wollen oder gar automatisiert E-Mails mit vorgefertigtem Inhalt an einen oder mehrere Empfänger gehen sollen. In diesem Fall benötigen Sie eine Möglichkeit, den Absender per Code einzustellen oder diesen zumindest gleich beim Zusammenstellen der Mail in der Anwendung festzulegen.

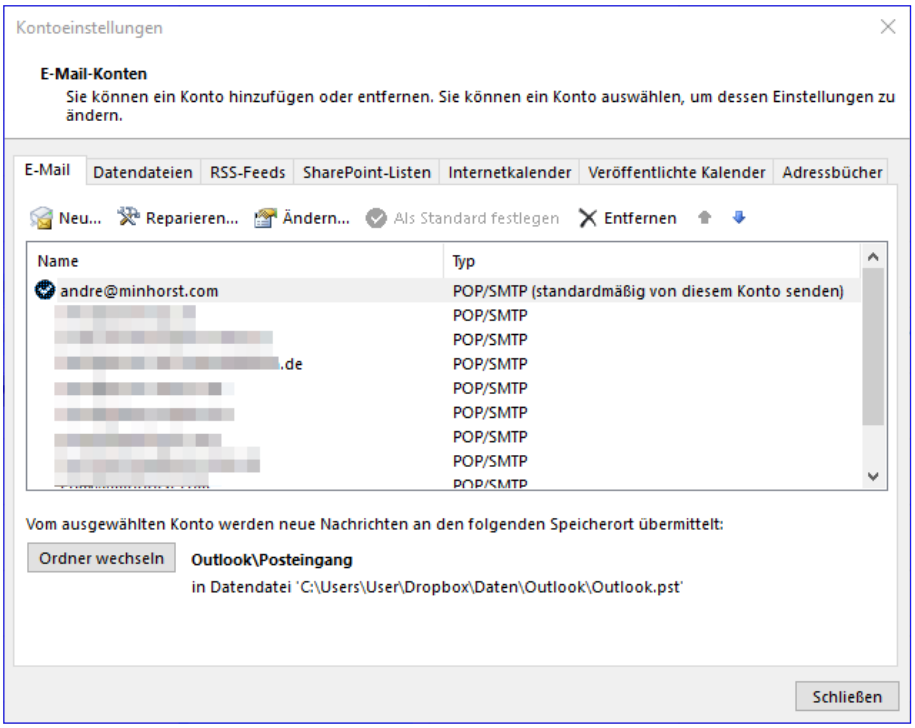


Bild 2: E-Mail-Konten unter Outlook

Absenderadresse festlegen

Wir werden uns zwei Varianten ansehen, um die Absenderadresse festzulegen. Bei der ersten geben wir die Absenderadresse einfach in ein Textfeld ein und versuchen dann beim Zusammenstellen der E-Mail, das betroffene Outlook-Konto zu ermitteln und als Absenderkonto einzustellen. Im zweiten Anlauf wollen wir dann vorab alle Outlook-Konten auslesen und die E-Mail-Adressen per Kombinationsfeld zur Auswahl bereitstellen.

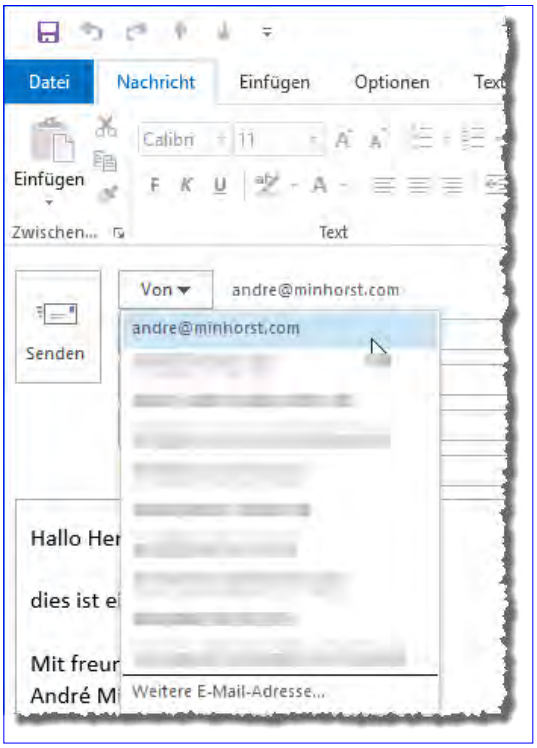


Bild 3: Auswahl des Absenders in der E-Mail

Wir bauen die Beispiellösung auf der Anwendung auf, die wir im Artikel **E-Mails mit Outlook verschicken** beschrieben haben. Dazu fügen wir zunächst ein Textfeld namens **txtAbsender** zum Fenster **MainWindow.xaml** hinzu:

```
<TextBox x:Name="txtAbsender" Grid.Column="1"></TextBox>
```

Damit dieses gleich beim Öffnen des Fensters gefüllt wird, fügen wir eine Zeile zur Methode **Window_Loaded** hinzu:

```
private void Window_Loaded(object sender, RoutedEventArgs e) {
    ...
    txtAbsender.Text = "info@amvshop.de";
    ...
}
```

E-Mails ohne Outlook versenden

Im Artikel »E-Mails mit Outlook verschicken« haben wir gezeigt, wie Sie von einer .NET-Anwendung aus die Daten für eine E-Mail zusammenstellen und diese dann per Outlook verschicken können. Was aber, wenn der Benutzer kein Outlook auf dem Rechner hat? Für diesen Fall bietet .NET eigene Bibliotheken. Damit können Sie E-Mails ohne Zugriff auf Outlook verschicken. Dieser Artikel zeigt, wie dies gelingt.

Wir wollen in diesem Artikel das gleiche Fenster verwenden, mit dem wir auch in der Beispiellösung zum Artikel **E-Mails mit Outlook verschicken** gearbeitet haben (siehe Bild 1). Einziger Unterschied: Die Schaltfläche **btnAnzeigen** fehlt, denn wir arbeiten ja nicht mit Outlook und können die Mail vor dem Versenden auch nicht mehr separat anzeigen. Dafür haben wir eine Schaltfläche namens Konfiguration hinzugefügt. Diese soll einen weiteren Dialog öffnen, der die Konfigurationsdaten für den Mailserver speichert.

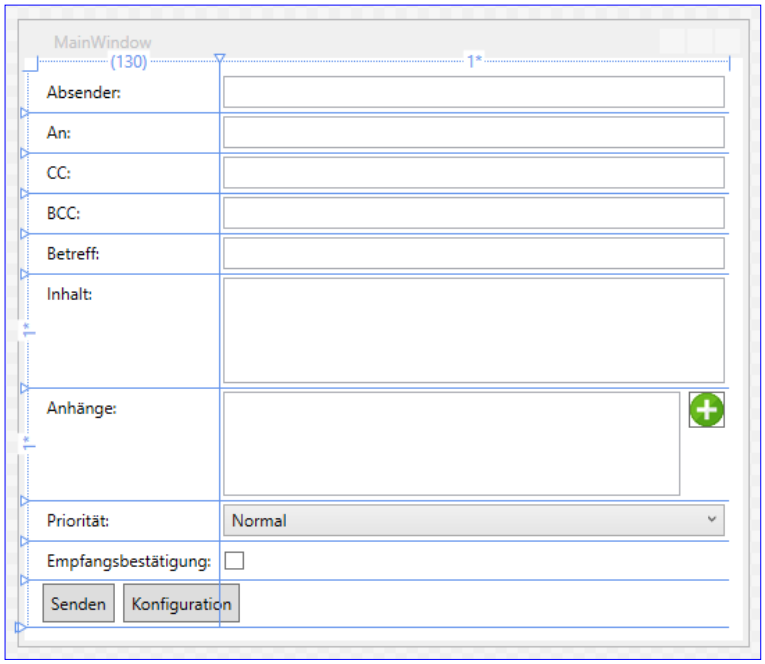


Bild 1: Fenster zur Eingabe der Mail-Daten

Für die Konfigurationsdaten legen wir einige Einträge im Bereich **Einstellungen** des aktuellen Projekts an, und zwar die folgenden (siehe Bild 2):

- **Host**: Name/IP des SMTP-Servers
- **Port**: Port des SMTP-Servers
- **Username**: Benutzername für die Anmeldung am SMTP-Server
- **Password**: Kennwort für die Anmeldung am SMTP-Server

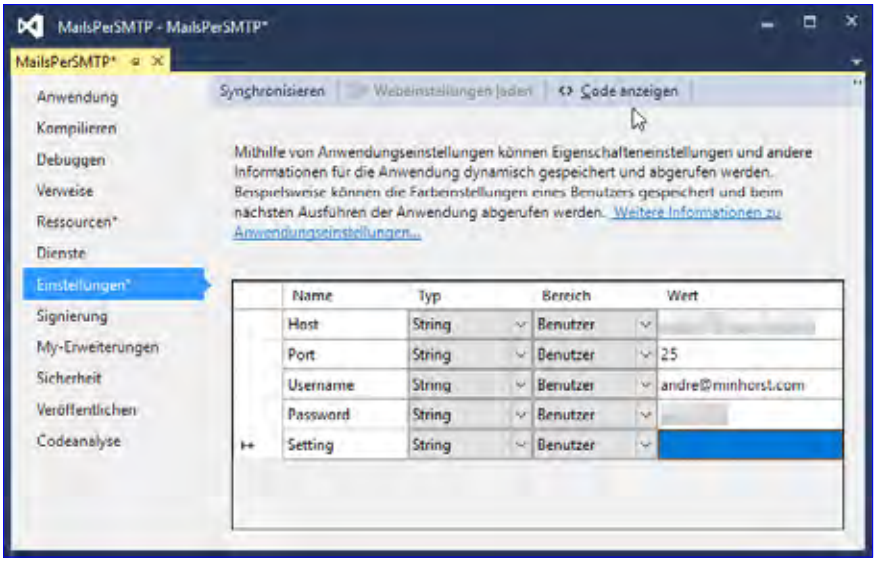


Bild 2: Anlegen der Konfigurationsdaten

Wenn Sie die Eigenschaften im Bereich **Einstellungen** angelegt haben und diese speichern, erstellt Visual Studio eine Klasse namens **Properties.Settings**. Diese können Sie dann initialisieren und damit direkt die angelegten Eigenschaften ansprechen, also etwa **setting.User-**

name – mehr dazu weiter unten. Die Ereignismethode **Window_Load** füllt die Felder **txtAbsender**, **txtAn**, **txtBetreff** und **txtInhalt** mit Beispieldaten. Dies war beim Testen des Beispiels hilfreich, Sie werden diese Felder jedoch auf andere Weise füllen wollen – vorausgesetzt, Sie wollen überhaupt Mails über ein Fenster versenden und nicht direkt etwa mit den Daten aus einer Datenbank.

Erstellen und Senden der E-Mail

Die durch die Schaltfläche **btnSenden** ausgelöste Ereignismethode stellt die E-Mail zusammen und verschickt diese dann. Da wir hier nicht Outlook als Mail-Client verwenden, sondern die Klasse **MailMessage**, benötigen wir etwas mehr Code für die Durchführung der Aufgabe. Das liegt vorrangig daran, dass wir die Daten des SMTP-Servers sowie die Benutzerdaten erst noch angeben müssen, die ja bei Benutzung von Outlook als Mail-Client bereits im jeweiligen Outlook-Konto gespeichert sind. Um die nachfolgend verwendeten Klassen nutzen zu können, benötigen wir noch Verweise auf die folgenden Namespaces:

```
using System.Net.Mail;
using System.Net;
```

Die Methode deklariert zunächst einige Variablen und initialisiert dann ein neues Objekt auf Basis der Klasse **MailMessage**:

```
private void btnSenden_Click(object sender, RoutedEventArgs e) {
    MailMessage mailMessage;
    MailAddress mailAddress;
    SmtpClient smtpClient;
    NetworkCredential networkCredentials;
    Attachment attachment;
    Properties.Settings settings;
    mailMessage = new MailMessage();
```

Dann beginnt die Methode, die Eigenschaften des neuen **MailMessage**-Objekts zu füllen. Als Absender geben wir für die Eigenschaft **From** ein Objekt des Typs **MailAddress** an, dem wir beim Initialisieren den Inhalt des Textfeldes **txtAn** übergeben:

```
    mailAddress = new MailAddress(txtAn.Text);
    mailMessage.From = mailAddress;
```

Die Empfängeradresse weisen wir mit der **Add**-Methode der Eigenschaft **To** zu. Betreff und Inhalt landen wie auch bei Outlook in den Eigenschaften **Subject** und **Body**:

```
    mailMessage.To.Add(txtAn.Text);
    mailMessage.Subject = txtBetreff.Text;
    mailMessage.Body = txtInhalt.Text;
```

Die Anlagen werden auch etwas anders zur E-Mail hinzugefügt wir unter Verwendung von Outlook. Wir können der **Attachments**-Auflistung hier nicht einfach den Namen der hinzuzufügenden Datei zuweisen, sondern müssen erst ein Objekt des Typs

Mit Ressourcen arbeiten

In einer .NET-Anwendung können Sie auch ohne Verwendung einer Datenbank Dateien und andere Daten ablegen. Dazu stellt Microsoft die sogenannten Ressourcen (englisch Resources) zur Verfügung. Dabei handelt es sich um die Möglichkeit, Dateien, Texte oder andere Daten entweder über die Benutzeroberfläche von Visual Studio zur Anwendung hinzuzufügen und sie per Code auszulesen und gegebenenfalls auch wieder ins Dateisystem zu kopieren. Im konkreten Fall wollen wir eine Word-Dokumentvorlage in der Anwendung vorhalten, um diese bei Bedarf ins Dateisystem zu kopieren und als Vorlage zu nutzen.

Beispiel Word-Dokumentvorlage

Anlass zu diesem Artikel ist ein weiterer Artikel namens **Briefe mit Word erstellen**, in dem wir auf Basis einer Word-Dokumentvorlage ein neues Word-Dokument erstellen wollen.

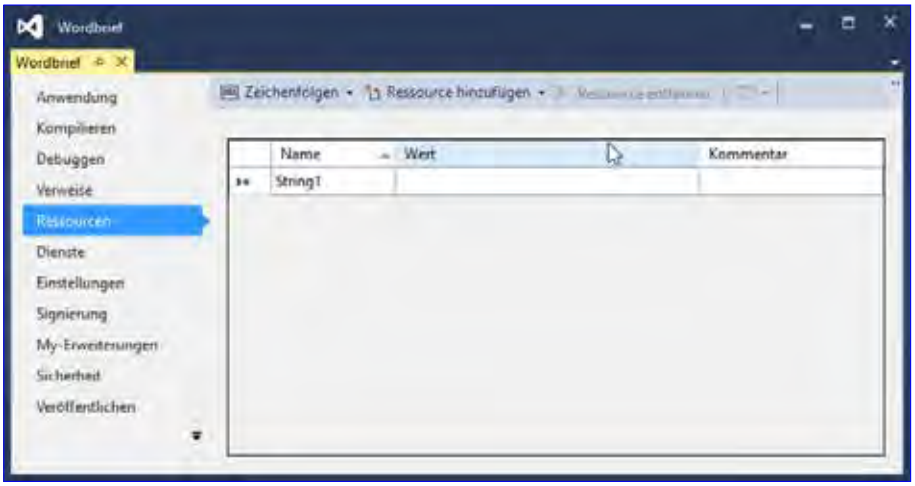


Bild 1: Der Dialog zum Verwalten der Ressourcen über Visual Studio

Für den Fall, dass der Benutzer die Anwendung frisch auf dem Rechner installiert hat, soll die Dokumentvorlage vor dem erstmaligen Erstellen von der Anwendung ins Dateisystem kopiert werden, damit der Code zum Erstellen des Word-Dokuments auf Basis dieser Vorlage auf die Vorlage zugreifen kann.

Ressourcen in Visual Studio verwalten

Der einfachste Weg, die Ressourcen zu verwalten, findet sich in der Benutzeroberfläche von Visual Studio. Wenn Sie ein Projekt angelegt haben, klicken Sie doppelt auf den Eintrag **My Project** im Projektmappen-Explorer, um die Eigenschaften der Anwendung anzuzeigen.

Diesen Bereich öffnen Sie auch mit dem Menübefehl **Projekt|Eigenschaften**. Hier wechseln Sie dann durch einen Klick auf den Eintrag **Ressourcen** in der linken Leiste zum gleichnamigen Bereich. In einem frischen Projekt ist die Liste der Ressourcen noch leer (siehe Bild 1).

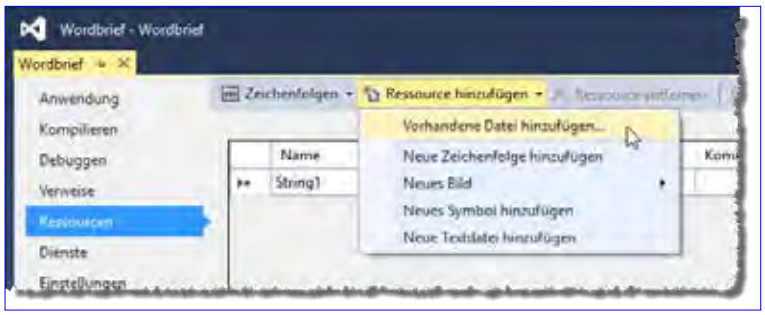


Bild 2: Hinzufügen einer Datei als Ressource

Um eine Datei als Ressource hinzuzufügen, klicken Sie auf den **Nach unten**-Pfeil der Schaltfläche **Ressource hinzufügen** im oberen Bereich und wählen im nun erscheinenden Menü den Eintrag **Vorhandene Datei hinzufügen...** aus (siehe Bild 2).

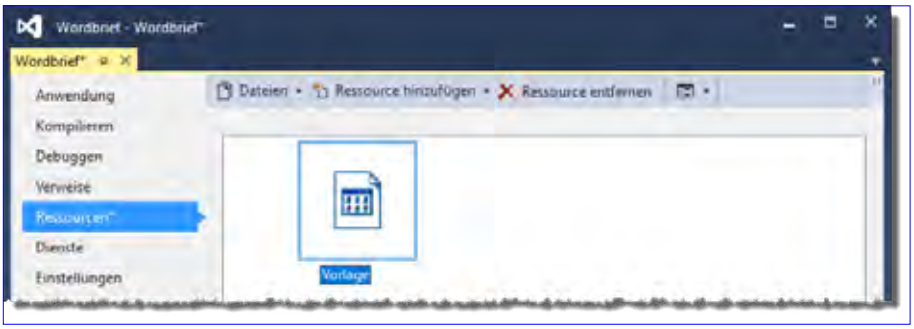


Bild 3: Anzeige der neuen Ressource in den Projekteigenschaften

Im folgenden Dialog **Vorhandene Datei zu Ressourcen hinzufügen** wählen Sie nun die hinzuzufügende Datei aus, in unserem Fall eine Datei namens **Vorlage.dotx**.

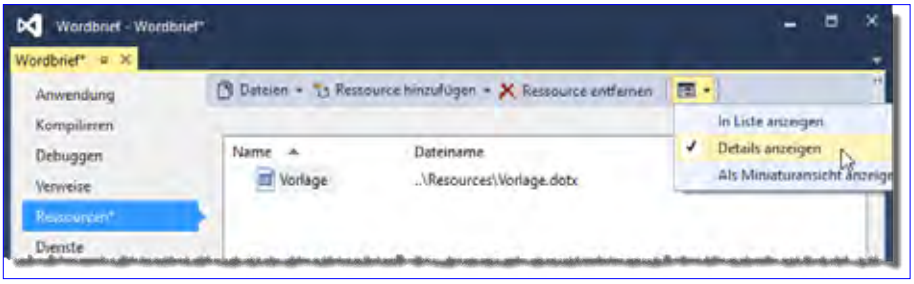


Bild 4: Detaillierte Liste der Ressourcen

Nach dem Hinzufügen zeigt der **Ressourcen**-Bereich die neue Ressource wie in Bild 3 an.

Diese Ansicht ist nicht besonders aussagekräftig, deshalb wechseln wir wie in Bild 4 über die Auswahl der Ansichten zur Ansicht **Details anzeigen**. Damit befindet sich die Ressource bereits in der Anwendung. Sie müssen nun noch die Änderungen speichern, am einfachsten mit der Tastenkombination **Strg + S**.

Ressource per Projektmappen-Explorer hinzufügen

Nach dem Hinzufügen der ersten Ressource zeigt der Projektmappen-Explorer einen Ordner für die Ressourcen an (siehe Bild 5). Dieser bietet auch ein Kontextmenü, über dessen Eintrag **Hinzufügen|Neues Element...** Sie den Dialog **Neues Element hinzufügen** öffnen. Hier können Sie allerdings nicht beliebige Dateien hinzufügen, sodass der Weg über den Ressourcen-Bereich in den Projekteigenschaften die einfachere Variante ist.

Zur Laufzeit mit Ressourcen arbeiten

Nun ist das primäre Ziel, diese Ressource zur Laufzeit aus der Anwendung ins Dateisystem zu kopieren. Dies wollen wir als Nächstes realisieren. Damit erschöpfen sich die Möglichkeiten auch: Die als Ressource

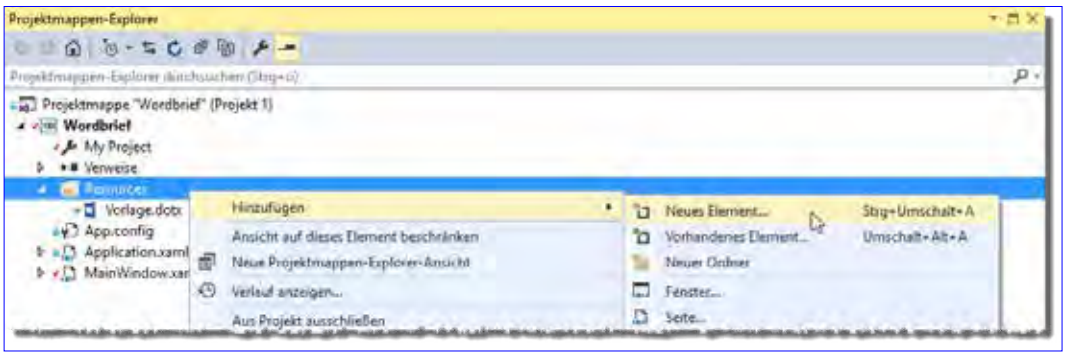


Bild 5: Ressource über den Projektmappen-Explorer hinzufügen

hinzugefügten Dateien werden nämlich beim Kompilieren in die **.exe**-Datei integriert. Sie können also nicht etwa zur Laufzeit weitere Elemente zu den Ressourcen einer Anwendung hinzufügen.

Schaltfläche zum Exportieren der Ressource

Für unsere erste Aufgabe fügen wir dem Fenster **MainWindow.xaml** eine Schaltfläche namens **btnRessourceExportieren** hinzu und legen für diese eine Ereignismethode an, die durch das Ereignis **Click** ausgelöst wird. Die Definition der Schaltfläche sieht so aus:

```
<Button x:Name="btnRessourceExportieren"
Click="btnRessourceExportieren_Click">
Ressource exportieren</Button>
```

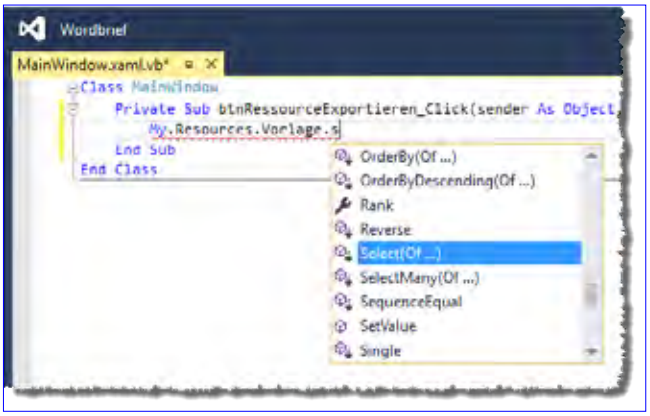


Bild 6: Zugriff auf die Ressource per Code

Wenn Sie nun zum Code-Editor für die Klasse **MainWindow.xaml.vb** wechseln, können Sie über **My.Resources** auf alle in den Ressourcen gespeicherten Elemente zugreifen, also auch auf unser Element mit der Bezeichnung **Vorlage**. Dieses bietet wiederum eine Reihe von Eigenschaften und Methoden an, unter denen sich aber leider nicht direkt eine Eigenschaft namens **SaveAs** findet (siehe Bild 6).

Wie bekommen wir die als Ressource gespeicherte Datei **Vorlage.dotx** nun dennoch in unser Dateisystem? Dazu brauchen wir die Methode lediglich wie folgt zu erweitern:

```
private void btnRessourceExportieren_Click(object sender, RoutedEventArgs e) {
    byte[] vorlage;
    vorlage = Properties.Resources.Vorlage;
    File.WriteAllBytes(AppDomain.CurrentDomain.BaseDirectory + "\\Vorlage.dotx", vorlage);
}
```

Die erste Zeile deklariert ein Byte-Array, die zweite weist diesem den Inhalt der Ressource zu, die wir mit **Properties.Resources.Vorlage** referenzieren. Die dritte speichert die Ressource unter dem Dateinamen **Vorlage.dotx** im aktuellen Verzeichnis der **.exe**-Anwendung.

Zielverzeichnis definieren

Nun wollen wir dem Benutzer noch die Möglichkeit geben, die Ressource in einem von ihm ausgewählten Verzeichnis zu speichern. Dazu legen wir eine zweite Schaltfläche namens **btnRessourceExportierenAls** an, für die wir die folgende Methode hinterlegen:

```
private void btnRessourceExportierenAls_Click(object sender, RoutedEventArgs e) {
    Byte[] vorlage;
    SaveFileDialog saveFileDialog;
```

Brief mit Word erstellen

Heute war es wieder soweit: Ich musste ein Anschreiben erstellen. Heutzutage geht zwar vieles per E-Mail, aber hier und da wird doch noch nach einem korrekten Anschreiben mit Briefkopf, Adresse, Ort und Datum, Betreff und dem Text selbst verlangt. Da dies so selten passiert, finde ich meist meine Wordvorlage für diesen Zweck nicht: Seit dem letzten Anschreiben kann es nämlich gut sein, dass ich einen neuen oder neu aufgesetzten Rechner habe und ich vergessen habe, meine Vorlagen vom alten Rechner auf den neuen zu übertragen. Wie schön wäre es doch, wenn man eine kleine .NET-Anwendung hätte, die einem ein paar Textfelder für die wichtigsten Daten bereitstellt und nach der Eingabe per Mausklick das gewünschte Dokument im .docx-Format anlegt! Wie das gelingt, zeigt Ihnen der vorliegende Artikel.

Wenn Sie ein Dokument wie oben beschrieben erstellen möchten, erledigen Sie das üblicherweise auf Basis einer Dokumentvorlage, die bereits einige wichtige Informationen enthält – zum Beispiel, wo die einzelnen Elemente wie Briefkopf, Anschrift, Ort und Datum, Betreff und Inhalt angelegt werden. Genau diese Dokumentvorlage benötigen wir auch für die Beispiellösung zu diesem Beitrag. Dumm nur, dass sich genau das Mitnehmen solcher Dokumente beim Aufsetzen eines neuen Rechners bei mir als Problem herausstellte und ich dann, statt die Vorlage auf dem alten Rechner zu suchen, lieber schnell von Hand ein neues Anschreiben zusammengesetzt habe. Also sollte die Lösung auch die Möglichkeit bieten, zumindest eine Dokumentvorlage zu speichern und diese dann bei Bedarf bereitzustellen. Wie dies gelingt, zeigen wir ausführlich im Artikel **Mit Resourcen arbeiten**.

Die grobe Vorgehensweise sieht so aus: Wir haben eine Anwendung, welche eine Word-Dokumentvorlage als Ressource enthält. Die Benutzeroberfläche stellt Textfelder bereits, mit denen Sie die im Anschreiben auszugebenden Informationen eintippen können. Nach der Eingabe können Sie per Mausklick einen Vorgang starten, bei dem die Anwendung die Dokumentvorlage im Dateisystem speichert (sofern diese noch nicht vorhanden ist) und ein neues Dokument auf Basis dieser Vorlage erstellt und mit den in der Anwendung eingegebenen Daten füllt. Das Dokument soll anschließend geöffnet werden, damit der Benutzer es beispielsweise ausdrucken kann. Alternativ wollen wir noch eine Schaltfläche anlegen, mit welcher der Benutzer direkt ein PDF-Dokument auf Basis des erstellten Word-Dokuments erzeugen kann.

Dokumentvorlage vorbereiten

Der erste Schritt ist das Vorbereiten einer Dokumentvorlage. Das heißt, dass wir in Microsoft Word ein neues Dokument anlegen, die gewünschten Bereiche inklusive Platzhalter definieren und beispielsweise den Briefkopf hinzufügen und das Dokument dann unter der Dateiendung **.dotx** speichern. Letzteres ist besonders wichtig, damit wir neue Dokumente auf Basis dieser Vorlage anlegen können. In welcher Form wollen wir die Platzhalter anlegen? Am einfachsten ist es, einfach entsprechende Texte in eckigen Klammern zu den jeweiligen Elementen hinzuzufügen, also beispielsweise **[Betreffzeile]**.

Vorlage nach DIN 5008

Wenn wir schon eine neue Vorlage anlegen, wollen wir diese auch halbwegs DIN-gerecht erstellen. Deshalb schauen wir uns dabei die aktuellen Vorgaben entsprechend DIN 5008 an, die etwa unter **www.din-5008-richtlinien.de** schön zusammenge-

fasst sind. Wir beginnen mit dem Anschriftenfeld, für das die DIN-Norm angibt, dass es 40 x 85 mm groß ist und einen Abstand von 45 mm zum oberen und 20 mm zum linken Rand aufweist (es gibt noch eine Variante mit 27 mm zum oberen Rand, aber wir wollen die Version für Briefumschläge mit Adressfenster nutzen).

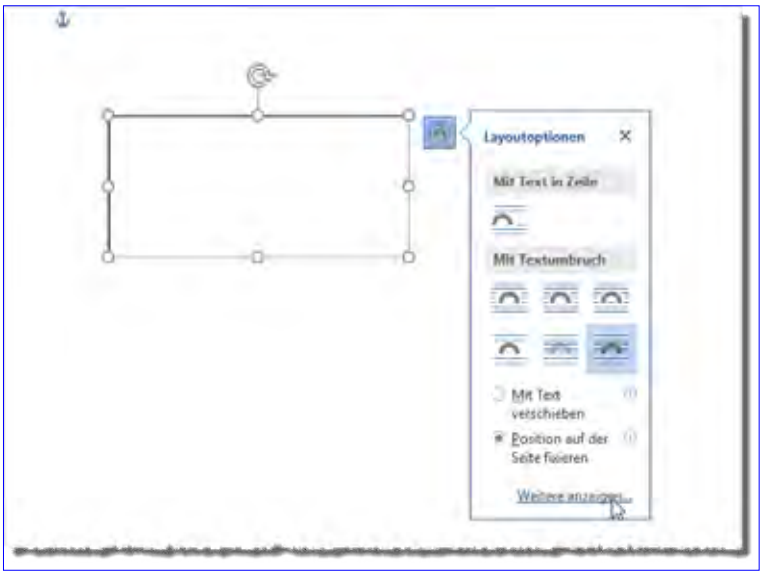


Bild 1: Anzeigen der Textfeld-Eigenschaften

Anschriftenfeld

Wir öffnen also Word und legen ein neues, leeres Dokument an. Für das Anschriftenfeld fügen Sie dann ein Textfeld hinzu. Dies erledigen wir mit dem Ribbon-Eintrag **Einfügen|Text|Textfeld|Textfeld** erstellen. Ziehen Sie nach Betätigung dieses Befehls einen Rahmen auf, der etwa den gewünschten Abmessungen entspricht. Klicken Sie dann auf die Schaltfläche, die rechts oberhalb vom neuen Textfeld erscheint, und wählen Sie unten den Befehl **Weitere anzeigen** aus (siehe Bild 1).

Bevor wir auf den Dialog **Layout** schauen, noch die Information, dass Word mitunter etwas verwirrend ist (Version 2016). Word zeigt nämlich standardmäßig den verfügbaren Bereich ohne Seitenränder an – zumindest ohne die oberen und unteren Seitenränder. Wenn man das weiß, kann man die Einstellungen im Layout-Dialog wie in Bild 2 ohne Rücksicht darauf vornehmen, dass die Position des Textfeldes in der Ansicht Drucklayout nicht so aussieht, wie sie später auf dem Papier erscheinen wird – also mit 2 cm Abstand zur Seite horizontal und mit 4,5 cm Abstand zur Seite vertikal.

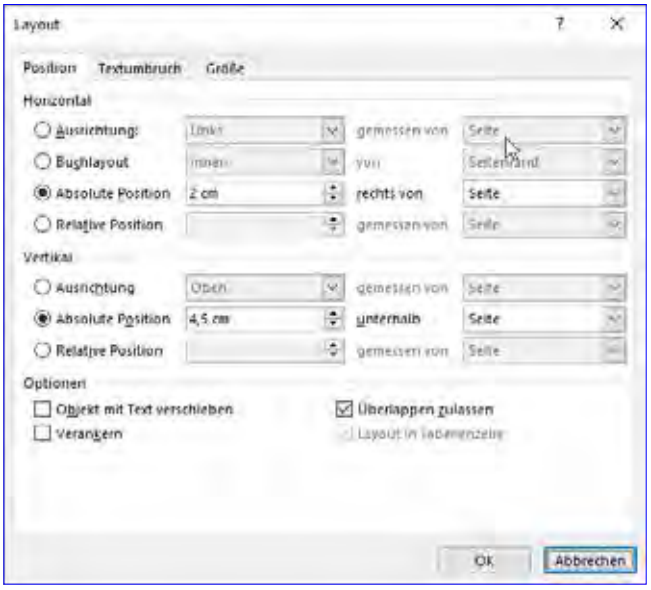


Bild 2: Position des Textfelds

Die Größe des Textfeldes stellen wir analog im Bereich Größe des gleichen Dialogs ein – und zwar wie von DIN 5008 vorgegeben auf 4 cm x 8,5 cm (siehe Bild 3). Die Größe können Sie allerdings auch über die beiden Steuerelemente in der Gruppe **Format|Größe** des Ribbons einstellen.

Bild 3: Größe des Textfelds

Schließlich weist das Textfeld noch einen Rahmen auf, den wir transparent machen wollen. Dazu wählen Sie beim markiertem Textfeld den Ribbon-Eintrag **Format|Formenarten|Formkontur|Keine Kontur** aus.

Bild 4: Einfügen einer Textmarke über das Ribbon

Ganz oben im Anschriftenfeld fügen wir nun eine Zeile mit der Rücksendeadresse ein, also der Adresse, an die der Brief zurückgesendet werden soll, wenn dieser nicht zugestellt werden konnte. Für diese Zeile legen Sie eine recht kleine Schriftgröße fest, zum Beispiel **8**. Darunter soll die eigentliche Anschrift landen – wie, erfahren Sie in den nächsten Abschnitten.

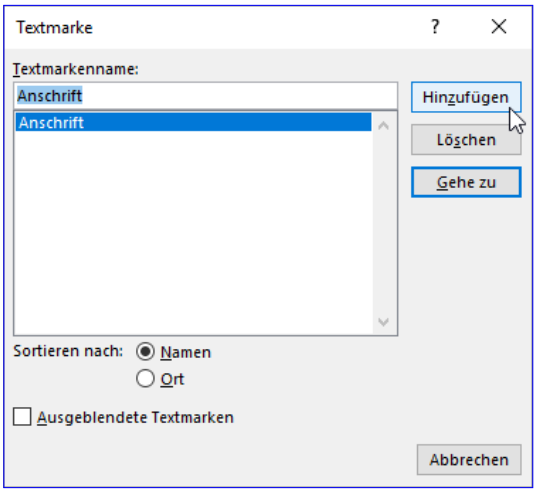


Bild 5: Einfügen einer Textmarke

Textmarke einfügen

Später wollen wir den Textfeldern Texte hinzufügen. Dazu hinterlegen wir in den Textfeldern jeweils eine Textmarke, die wir dann später per Code leicht auffinden und durch die gewünschten Texte ersetzen können. Um die Textmarke hinzuzufügen, platzieren Sie die Einfügemarke im Textfeld für die Anschrift gleich unter der Rücksendeadresse und stellen die Schriftart auf die gewünschte Größe ein, beispielsweise auf **10**. Danach fügen wir die Textmarke ein. Dies erledigen wir mit dem Ribbon-Befehl **Einfügen|Links|Textmarke** (siehe Bild 4).

Dies öffnet den Dialog aus Bild 5, mit dem wir den Namen der neuen Textmarke eingeben und diese dann mit der Schaltfläche **Hinzufügen** an der Position der Einfügemarke einfügen. Im Dokument ist die Textmarke nun zunächst nicht zu erkennen. Das Vorhandensein dokumentiert lediglich der neue Eintrag im Dialog **Textmarke**.

Zur Sicherheit wollen wir die Textmarke jedoch noch sichtbar machen. Dazu öffnen Sie über das Ribbon den Dialog **Word-Optionen** und klicken dort links auf den Eintrag **Erweitert**. Im rechten Bereich finden Sie dann unter Dokumentinhalt anzeigen die Option **Textmarken anzeigen**, die Sie nun aktivieren (siehe Bild 6).

Wenn Sie den Dialog nun mit einem Klick auf die Schaltfläche

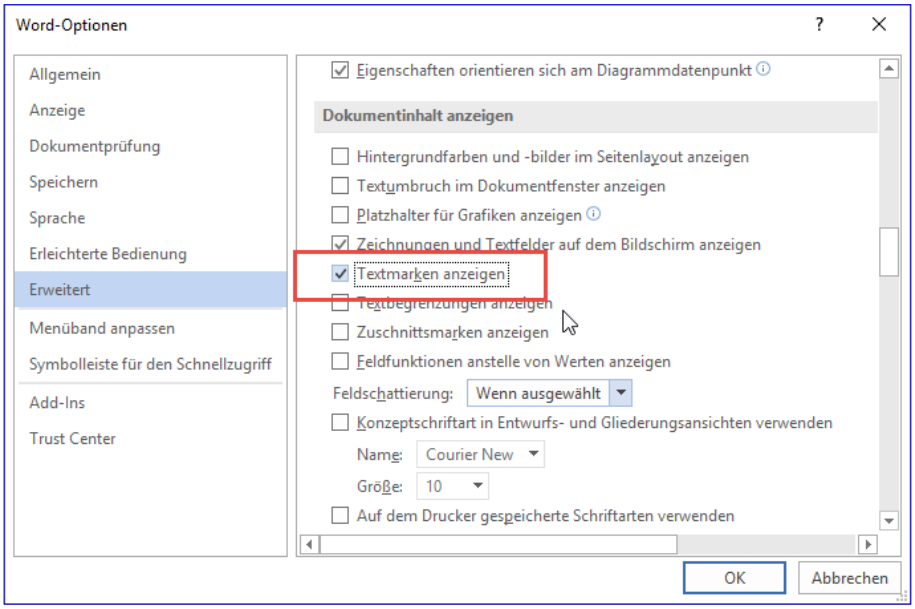


Bild 6: Textmarken einblenden

OK schließen, finden Sie im Textfeld für die Anschrift die Textmarke wie in Bild 7 vor.

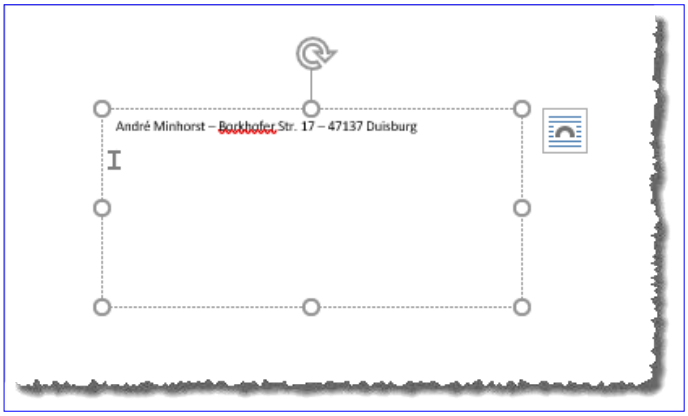


Bild 7: Markierung für eine Textmarke

Wenn Sie später alle Textfelder sichtbar machen wollen, können Sie einfach in einen leeren Bereich des Dokuments klicken und alle Elemente mit der Tastenkombination **Strg + A** markieren.

Das Textfeld können Sie nun gleich kopieren und als Basis für das Feld für die Betreffzeile nutzen – so brauchen Sie den Rahmen nicht wieder wie bei einem neuen Textfeld auf transparent einzustellen. Die Betreffzeile soll zwei Zeilen unterhalb des Anschriftenfeldes liegen, was wir als ca. 10 mm interpretieren. Also stellen wir den Abstand vom oberen Rand auf 10,5 cm ein, den Abstand vom linken Rand auf 2 cm und die Größe auf 1 x 12 cm. Löschen Sie den mitkopierten Text, stellen Sie die Schriftgröße auf 10 ein und fügen Sie auch diesem Textfeld einen Platzhalter hinzu – diesmal mit der Bezeichnung **Betreff**.

Rechts oben wollen wir den Briefkopf unterbringen. Dafür legen wir ein Textfeld mit einem Abstand von 2 cm vom rechten Rand und 2,5 cm vom oberen Rand und mit der Größe 5 x 5 cm an. Dazu müssen wir ein wenig rechnen, denn wir können nur Abstände von oben und links eintragen. Ein DIN A4-Blatt hat eine Breite von 19,7 cm, also müssen wir 2 cm für den Abstand von rechts und 5 cm für die Breite abziehen und landen bei einem Abstand von 12,7 cm vom linken Rand. Den Briefkopf füllen Sie mit den Absenderdaten aus und mit weiteren Daten, die Sie angeben möchten – Telefon, E-Mail, Webseite, Bankverbindung, Steuernummern et cetera.

Das Datum des Anschreibens fügen wir in einem Textfeld ein, das die Position 12,7 cm vom linken Rand und 9 cm vom oberen Rand einnimmt und eine Größe von 5 x 1 cm aufweist. Das Textfeld füllen wir mit einer weiteren Textmarke, diesmal mit der Bezeichnung **Datum** in Schriftgröße 10.

Das letzte Textfeld soll den Inhalt des Anschreibens aufnehmen. Es hat einen Abstand von 2 cm zum linken Rand und 13 cm zum oberen Rand und eine Größe von 15,7 x 12 cm. Hier fügen wir die Textmarke mit der Bezeichnung **Inhalt** ein.

An dieser Stelle der Hinweis, dass wir hier nur in Textfeldern schreiben und daher unabhängig von den Einstellungen der Seitenränder des Dokuments sind. Damit erhalten wir allerdings die Einschränkung, dass wir auch nur den Platz des dafür vorgesehenen Textfeldes für den Inhalt des Anschreibens haben. Für mehrseitige Dokumente müssten wir allerdings auch noch etwas mehr Aufwand betreiben – das soll jedoch nicht Thema dieses Artikels sein. Das Dokument sieht schließlich wie in Bild 8 aus. Schließlich speichern Sie die Word-Datei unter dem Namen **Vorlage.dotx**, wozu Sie den entsprechenden Dateityp beim Speichern auswählen müssen.

Vorlage wieder öffnen

Wenn Sie die Dokumentvorlage nach dem Schließen nochmals öffnen wollen, um diese beispielsweise nochmals anzupassen, können Sie dies nicht einfach per Doppelklick auf die Datei **Vorlage.dotx** erledigen. Dies öffnet nämlich nicht die Dokument-

vorlge, sondern erzeugt ein neues Dokument auf Basis dieser Vorlage.

Dieses sieht dann zwar genauso aus wie die Vorlage selbst, aber wenn Sie diese speichern, stellen Sie fest, dass es sich hier um ein einfaches Word-Dokument handelt.

Wenn Sie die Vorlagendatei öffnen wollen, starten Sie zunächst Word und öffnen die Vorlagendatei dann über den **Datei öffnen**-Dialog von Word.

Steuerelemente zum Eingeben der Briefdaten

Damit kommen wir zurück zum Visual Studio-Projekt. Diesem fügen wir nun zunächst die fertige Word-Vorlage als Ressource hinzu. Dazu öffnen Sie die Projekteigenschaften, wählen den Eintrag **Ressourcen** und legen die Word-Dokumentvorlage als neue Ressource an. Details dazu finden Sie im Artikel **Mit Ressourcen arbeiten**.

Wir machen es auf die Schnelle so, dass wir einfach die Datei **Vorlage.dotx** aus dem Windows Explorer in die Liste der Ressourcen hineinziehen (siehe Bild 9).

Sie können natürlich auch noch weitere Vorlagen anlegen, die beispielsweise für andere Familienmitglieder genutzt werden oder für geschäftliche Zwecke.

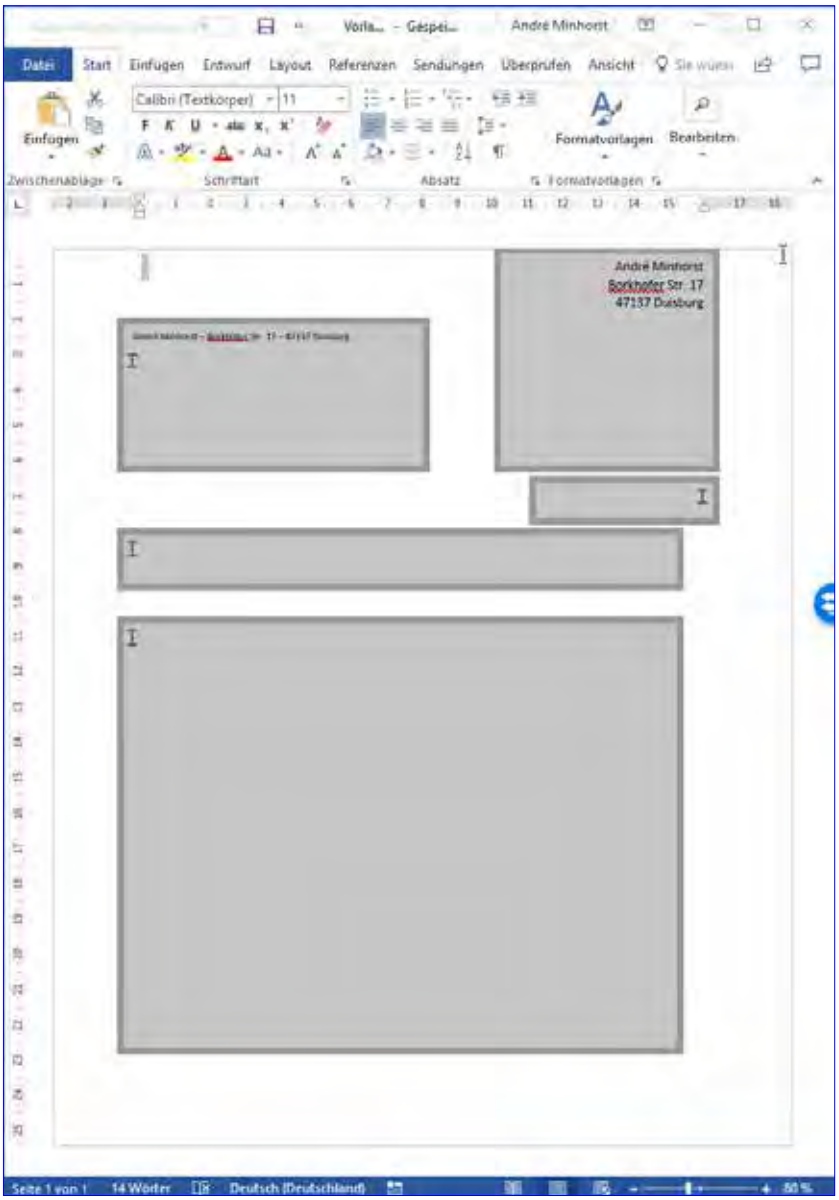


Bild 8: Anordnung der Textfelder

Anzeige und Auswahl der Vorlagen

Ganz oben im Fenster wollen wir ein Kombinationsfeld zur Auswahl der Vorlage einfügen. Das Label und das Kombinationsfeld definieren wir wie folgt:

```
<ComboBox x:Name="cboVorlagen" Grid.RowSpan="2" Grid.Column="1"/>
<Button x:Name="btnGewaehlteRessourcenExportieren" Click="btnGewaehlteRessourcenExportieren_Click"
    Grid.Row="8">Gewählte Ressource exportieren</Button>
```

Damit das Kombinationsfeld mit den verfügbaren Vorlagen gefüllt wird, legen wir für das Window-Element die Ereigniseigenschaft **Loaded** an:

```
<Window x:Class="MainWindow" ...
Title="MainWindow" Height="450"
Width="525" Loaded="Window_Loaded">
```

Für diese hinterlegen wir die folgende Methode:

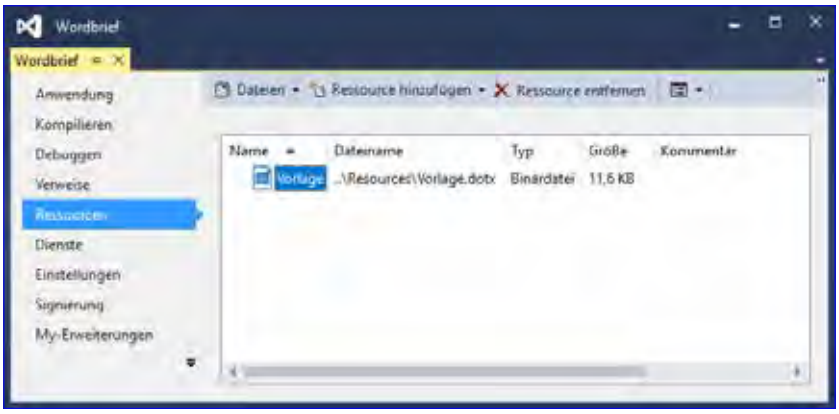
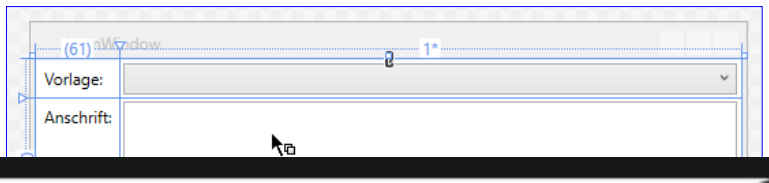


Bild 9: Voraussetzung: Eine Word-Dokumentvorlage als Ressource

```
private void Window_Loaded(object sender, RoutedEventArgs e) {
    ResourceManager resourceManager;
    ResourceSet resourceSet;
    string strResource;
    resourceManager = Properties.Resources.ResourceManager;
    resourceSet = resourceManager.GetResourceSet(CultureInfo.CurrentCulture, true, true);
    cboVorlagen.Items.Add("<Auswählen>");
    foreach (DictionaryEntry dictionaryEntry in resourceSet) {
        strResource = dictionaryEntry.Key.ToString();
        cboVorlagen.Items.Add(strResource);
    }
    cboVorlagen.SelectedIndex = 0;
}
```



Die Methode erstellt ein **ResourceManager**-Objekt auf Basis der in der Anwendung befindli-

Die Leseprobe dieses Artikels ist hier zu Ende.
Willst Du mehr?
ZUM SHOP
2.400 Seiten Know-how rund um die Datenbankentwicklung mit .NET
KOMPENDIUM
DATENBANK ENTWICKLER
KOMPENDIUM
André Minhorst Verlag
Spare 20 EUR mit Code dbew20

AUSGABE 01/2018

DATENBANK ENTWICKLER

MAGAZIN FÜR DIE DATENBANKENTWICKLUNG MIT VISUAL STUDIO FÜR DESKTOP, WEB UND CO.

TOP-THEMEN:

WPF-CONTROLS	Das DatePicker-Steuerelement	SEITE 3
WPF-CONTROLS	Das ListView-Steuerelement	SEITE 11
VB-BASICS	Mit Klassen programmieren	SEITE 23
DATENZUGRIFF	EDM: Der Code First-Ansatz	SEITE 36
LÖSUNGEN	Adressen verwalten	SEITE 55

Das DatePicker-Steuerelement

Für die Eingabe von Datumsangaben gibt es unter WPF ein spezielles Steuerelement – das sogenannte DatePicker-Control. Es ist recht schlicht gehalten, bietet aber eine gute und zuverlässige Möglichkeit, Datumsangaben schnell auszuwählen. Da es ein spezialisiertes Steuerelement ist, offeriert es jedoch auch einige individuelle Attribute, die Sie kennen sollten, um das Element effizient zu nutzen. Anderenfalls kommt es schnell zu Problemen ...

In einer Datenbankanwendung ist die Wahrscheinlichkeit sehr hoch, dass Sie früher oder später Datumsangaben eingeben müssen. Unter Access wurde für Felder, die auf einem Datumsfeld basierten, ein Textfeld angelegt, das beim Fokuserhalt eine Schaltfläche zum Öffnen eines kleinen Popups zur Datumsauswahl präsentierte. Bei dem **DatePicker**-Steuerelement von WPF ist diese kleine Schaltfläche immer sichtbar, sodass der Benutzer direkt weiß, dass er bei der Datumseingabe Unterstützung findet. Ein **DatePicker**-Element ist schnell hinzufügt – entweder über die ToolBox, wo es unter **Alle WPF-Steuerelemente** zu finden ist oder durch das Hinzufügen eines neuen Elements des Typs **DatePicker** zum XAML-Code – im einfachsten Fall wie folgt:

```
<DatePicker />
```

Das Steuerelement sieht dann im Entwurf wie in Bild 1 aus.

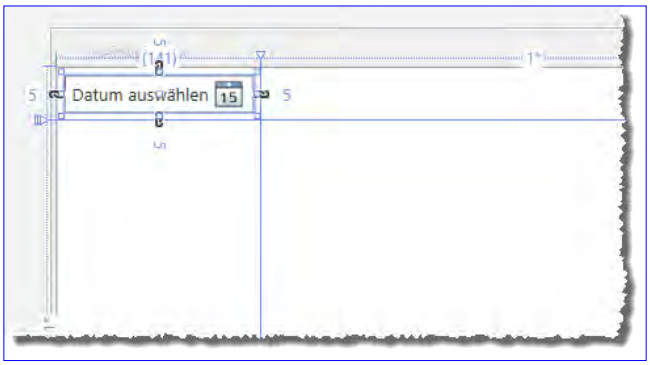


Bild 1: Das **DatePicker**-Element in der Entwurfsansicht

Wenn Sie die Anwendung starten, liefert das **DatePicker**-Steuerelement den Text **Datum auswählen**. Ein Klick auf die Schaltfläche rechts neben dem Eingabefeld zeigt dann ein Popup mit der Ansicht des aktuellen Monats an (siehe Bild 2).

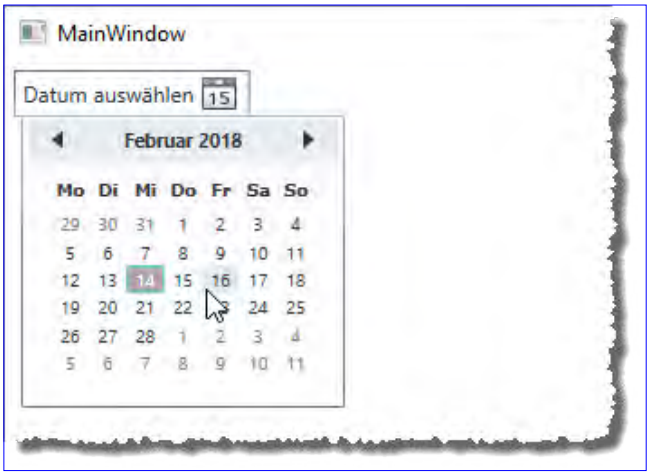


Bild 2: Das **DatePicker**-Element in Aktion

Besondere Eigenschaften des DatePicker-Elements

Das **DatePicker**-Element liefert einige Eigenschaften, die für die Programmierung interessant sind.

- **SelectedDateChanged**: Diese Ereignis wird ausgelöst, wenn sich der Wert des Datums ändert.
- **IsDropDownOpen**: Gibt den aktuellen Zustand des Kalender-Popups wieder und erlaubt auch die Einstellung der Eigenschaft.

- **SelectedDateFormat**: Gibt das angezeigte Datumsformat aus.
- **CalendarOpened**: Wird ausgelöst, wenn der Kalender geöffnet wird.
- **CalendarClosed**: Wird ausgelöst, wenn der Kalender geschlossen wird.
- **DateValidationError**: Wird ausgelöst, wenn ein ungültiges Datum eingegeben wird.

Ereignis beim Ändern des Datums

Wenn der Benutzer das Datum im **DatePicker**-Element anpasst, löst er damit das Ereignis **SelectedDateChanged** aus, welches wir mit der folgenden Methode implementieren können. Das neue Datum wird dann in einem Meldungsfenster angezeigt:

```
Private Sub dpDatepicker_SelectedDateChanged(sender As Object, e As SelectionChangedEventArgs)
    Dim datePicker As DatePicker
    datePicker = sender
    MessageBox.Show("Das neue Datum lautet: " + datePicker.Text)
End Sub
```

Zuvor legen wir das Attribut für das **DatePicker**-Element wie folgt an:

```
<DatePicker x:Name="dpDatepicker" SelectedDateChanged="DatePicker_SelectedDateChanged"></DatePicker>
```

Datumsformat ermitteln

Die **MessageBox** der obigen Ereignismethode können wir noch erweitern. Sie soll nun auch noch das Format des angezeigten Datums ausgeben:

```
MessageBox.Show("Das neue Datum lautet: " + datePicker.Text + vbCrLf + "und hat das Format: " + datePicker.SelectedDate-
Format.ToString)
```

Mögliche Werte sind **Short** und **Long** – hier mit Beispielen:

- **Short**: **15.02.2018**
- **Long**: **Donnerstag, 15. Februar 2018**

Kalender per Code aufklappen

Vielleicht möchten Sie den Kalender automatisch aufklappen – zum Beispiel, wenn sich das Fenster öffnet oder wenn der Benutzer ein ungültiges Datum eingegeben hat. Dann können Sie die Eigenschaft **IsDropDownOpen** nutzen. Diese liefert sowohl den aktuellen Zustand als auch die Möglichkeit, den Zustand durch Zuweisen eines **Boolean**-Wertes zu ändern. Die Schaltfläche, die wir im Fenster neben dem **DatePicker** untergebracht haben, macht genau das – sie klappt den Kalender auf:

```
Private Sub btnKalenderAnzeigen_Click(sender As Object, e As RoutedEventArgs)
    dpCatepicker.IsDropDownOpen = True
End Sub
```

Ereignisse bei Öffnen und Schließen des Kalenders

Die beiden Ereignisse **CalendarOpened** und **CalendarClosed** werden ausgelöst, wenn der Kalender angezeigt oder ausgeblendet wird. Sie werden wie folgt zum Element hinzugefügt:

```
<DatePicker x:Name="dpDatepicker" SelectedDateChanged="dpDatepicker_SelectedDateChanged" CalendarOpened="dpDatepicker_CalendarOpened" CalendarClosed="dpDatepicker_CalendarClosed"></DatePicker>
```

Die beiden Ereignisse programmieren wir wie folgt, um den aktuellen Zustand des Kalenders im Ausgabe-Bereich auszugeben:

```
Private Sub dpDatepicker_CalendarOpened(sender As Object, e As RoutedEventArgs)
    Debug.WriteLine("Der Kalender wurde geöffnet.")
End Sub

Private Sub dpDatepicker_CalendarClosed(sender As Object, e As RoutedEventArgs)
    Debug.WriteLine("Der Kalender wurde geschlossen.")
End Sub
```

Ungültige Eingaben abfangen

Wenn Sie in der Standardkonfiguration ein ungültiges Datum wie etwa **30.2.2018** eingeben, wird das Textfeld ohne Kommentar wieder geleert. Das überhaupt etwas geschieht, erfahren Sie im Ausgabe-Bereich – hier erscheint die folgende Meldung:

```
Ausnahme ausgelöst: "System.FormatException" in mscorlib.dll
```

Diese Ausnahme können Sie mit einer herkömmlichen Fehlerbehandlung (**Try...Catch**) nicht abfangen. Allerdings bietet das **DatePicker**-Element eine eigene Ereignismethode an, um auf solche Fehleingaben zu reagieren, und zwar das Ereignis **DateValidationError**:

```
<DatePicker x:Name="dpDatepicker" ... DateValidationError="dpDatepicker_DateValidationError"></DatePicker>
```

Mit der ersten Version der Ereignismethode zeigen wir nur die Fehlerart an, in diesem Fall **System.FormatException**:

```
Private Sub dpDatepicker_DateValidationError(sender As Object, e As DatePickerDateValidationErrorEventArgs)
    MessageBox.Show("Fehler: " + e.Exception.GetType.ToString)
End Sub
```

Geben wir in der Methode den Inhalt der Eigenschaft **Message** aus, erhalten wir die Meldung aus Bild 3.

Bestimmten Datumsbereich anzeigen

Normalerweise zeigt der Kalender folgenden Monat an:

- den aktuellen Monat, wenn noch kein Wert für das Steuerelement ausgewählt wurde oder
- den Monat, in dem sich der im Steuerelement angegebene Tag befindet.

Bild 3: Fehler bei ungültigem Datumsformat

Sie können dies jedoch auch variieren, indem Sie die Eigenschaften **DisplayDateStart** und **DisplayDateEnd** nutzen. Damit können Sie nicht nur den gewünschten Monat einstellen, sondern auch kürzere Bereiche. Das folgende Beispiel zeigt nur eine einzige Woche wie in Bild 4 an:

```
Private Sub btnKalenderMitAnderemDatum_Click( _
        sender As Object, e As RoutedEventArgs)
    With dpDatepicker
        .DisplayDateStart = "22.01.2018"
        .DisplayDateEnd = "28.01.2018"
        .SelectedDate = "23.01.2018"
        .IsDropDownOpen = True
    End With
End Sub
```

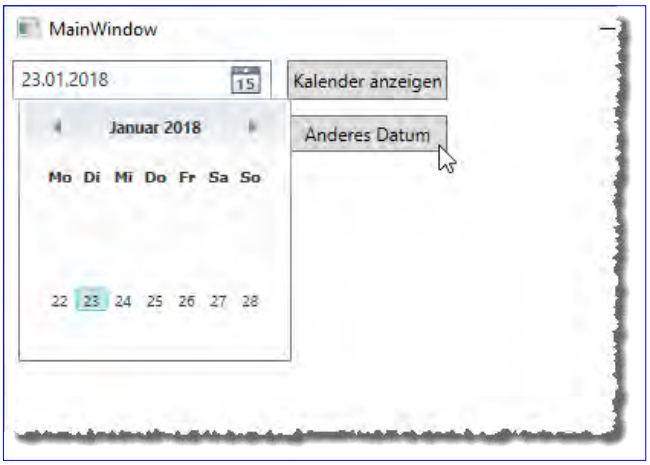


Bild 4: Anzeige nur einer einzigen Woche

Sie können jedoch maximal einen Zeitraum von sechs Wochen anzeigen.

Bestimmte Tage sperren

Sie können auch bestimmte Tage oder auch Datumsbereiche als gesperrt markieren. Wenn Sie etwa den **23.1.2018** sperren wollen, weil Sie dann Geburtstag haben, öffnen Sie den Kalender wie folgt:

```
Private Sub btnTageDeaktivieren_Click(sender As Object, e As RoutedEventArgs)
    With dpDatepicker
        .BlackoutDates.Add(New CalendarDateRange(New DateTime(2018, 1, 23)))
        .DisplayDateStart = "01.01.2018"
        .DisplayDateEnd = "31.01.2018"
        .IsDropDownOpen = True
    End With
End Sub
```

Das ListView-Steuerelement

Es gibt unter WPF sechs Steuerelemente, die eine Auswahl untergeordnete Elemente ermöglichen – ComboBox, ListBox, ListView, DataGrid, TabControl und Ribbon. In diesem Artikel schauen wir uns das ListView-Steuerelement an, das gegebenüber dem ListBox-Element einige Möglichkeiten zur einfacheren Darstellung der enthaltenen Daten liefert, und zwar über die View-Eigenschaft. Im Vergleich zur ListBox bietet die ListView außerdem standardmäßig die erweiterte Auswahl an – Sie können also direkt mehrere Einträge statt nur einem einzigen selektieren. Dieser Artikel zeigt, wie Sie das ListView-Steuerelement zur Anzeige von Daten aus einer Collection von Entitäten anzeigen können.

Das **ListView**-Steuerelement bietet eine ziemlich einfache Möglichkeit, die Daten etwa aus einer **ObservableCollection** zur Auswahl anzubieten. Für die Anzeige einiger Daten am Beispiel von Adressdaten haben wir die folgende Klasse für das Fenster **MainWindow.cs** eines frischen WPF-Projekts angelegt. Dieses hält eine **ObservableCollection** mit Elementen des Typs **Adresse** namens **Adressen** als öffentliche Eigenschaft bereit und füllt diese Collection in der Konstruktor-Methode mit einigen Elementen:

```
Class MainWindow
    Public Property Adressen As ObservableCollection(Of Adresse)
    Public Sub New()
        InitializeComponent()
        Adressen = New ObservableCollection(Of Adresse)
        Adressen.Add(New Adresse With {.Anrede = "Herr", .EMail = "andre@minhorst.com", .Firma = "André Minhorst Verlag", _
            .ID = 1, .Land = "Deutschland", .Nachname = "Minhorst", .Ort = "Duisburg", .PLZ = "47137", _
            .Strasse = "Borkhofer Str. 17", .Telefon = "0123/123456", .Vorname = "André"})
        '...weitere Adressen
        DataContext = Me
    End Sub
End Class
```

Die Klasse **Adresse.vb** sieht etwa wie folgt aus:

```
Public Class Adresse
    Public Property ID As Integer
    Public Property Firma As String
    Public Property Anrede As String
    Public Property Vorname As String
    Public Property Nachname As String
    Public Property Strasse As String
```

```
    Public Property PLZ As String
    Public Property Ort As String
    Public Property Land As String
    Public Property EMail As String
    Public Property Telefon As String
End Class
```

Damit können wir schon das **Window**-Element mit dem **ListView**-Element bestücken. Damit dieses die Daten der Auflistung in der **ObservableCollection** anzeigt, benötigen wir ein paar Unterelemente. Als Erstes stellen wir jedoch für das **ListView**-Element die Datenherkunft ein, und zwar mit dem Attribut **ItemsSource**.

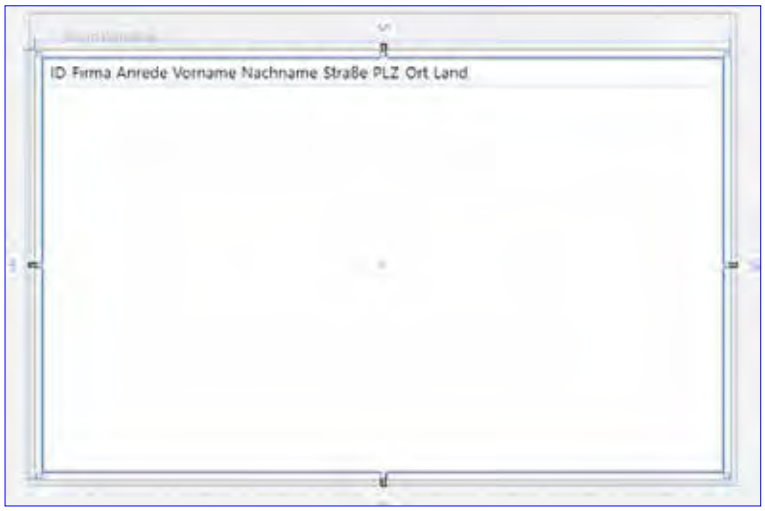


Bild 1: Entwurf des **ListView**-Elements

Dieses legen wir auf den Wert **{Binding Adressen}** ein, was eine Bindung zu der mit der öffentlichen Eigenschaft **Adressen** der Code behind-Klasse erzeugt. Dann fügen wir diesem über ein Property Element die Eigenschaft **View** hinzu (**<ListView.View>**). Dieses erhält schließlich das Element **GridView** und die untergeordneten Spaltenbeschreibungen **GridViewColumn**.

In diesen Elementen legen wir mit der Eigenschaft **Header** die Spaltenüberschriften und mit der Eigenschaft **DisplayMemberBinding** und dem Wert **{Binding <Feldname>}** den Inhalt des **ListView**-Elements fest:

```
<Window x:Class="MainWindow" ...        Title="MainWindow" Height="350" Width="525">
    <Grid>
        <ListView ItemsSource="{Binding Adressen}" Margin="5">
            <ListView.View>
                <GridView>
                    <GridViewColumn Header="ID" DisplayMemberBinding="{Binding ID}" />
                    <GridViewColumn Header="Firma" DisplayMemberBinding="{Binding Firma}" />
                    <GridViewColumn Header="Anrede" DisplayMemberBinding="{Binding Anrede}" />
                    <GridViewColumn Header="Vorname" DisplayMemberBinding="{Binding Vorname}" />
                    <GridViewColumn Header="Nachname" DisplayMemberBinding="{Binding Nachname}" />
                    <GridViewColumn Header="Straße" DisplayMemberBinding="{Binding Strasse}" />
                    <GridViewColumn Header="PLZ" DisplayMemberBinding="{Binding PLZ}" />
                    <GridViewColumn Header="Ort" DisplayMemberBinding="{Binding Ort}" />
                    <GridViewColumn Header="Land" DisplayMemberBinding="{Binding Land}" />
                </GridView>
            </ListView.View>
        </ListView>
    </Grid>
</Window>
```

Das Ergebnis sieht dann im Entwurf zunächst wie in Bild 1 aus.

Starten Sie nun die Anwendung, erhalten Sie bereits die mit den Adressen gefüllte Liste aus Bild 2.

Hier ein kurzer Hinweis darauf, dass wir für das Anlegen der **GridViewColumn**-Elemente die Standardeigenschaft des **GridView**-Elements nutzen, nämlich **Columns**. Ausgeschrieben sähe der Code in gekürzter Form wie folgt aus:

```
<GridView>
    <GridView.Columns>
        <GridViewColumn Header="ID"
DisplayMemberBinding="{Binding ID}" />
    </GridView.Columns>
</GridView>
```

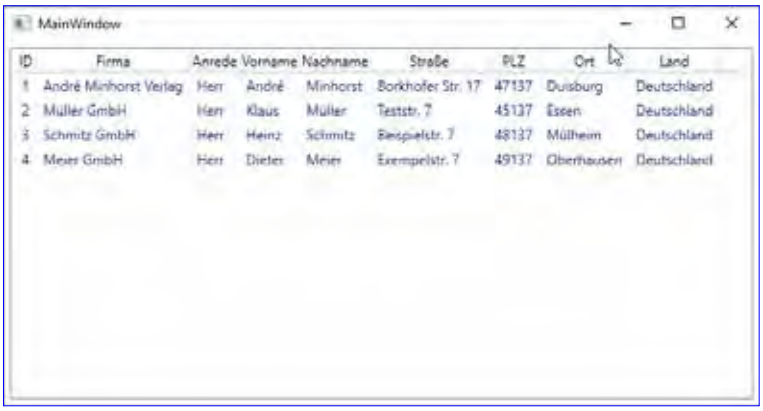


Bild 2: Das **ListView**-Element in Aktion

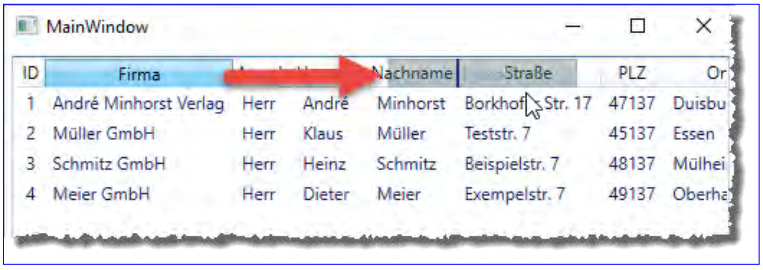


Bild 3: Änderung der Sortierung der Spalten

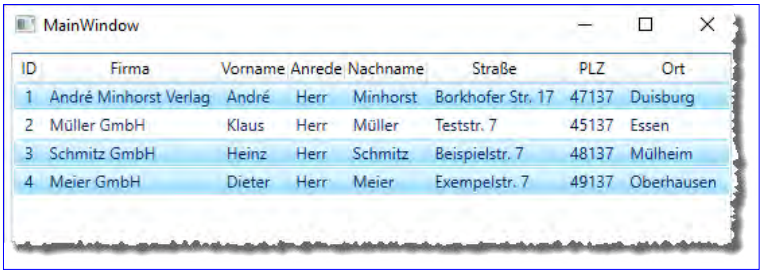


Bild 4: Erweiterte Mehrfachmarkierung

Eigenschaften des ListView-Steuerelements

Damit können wir uns den Eigenschaften und Möglichkeiten des **ListView**-Steuerelements zuwenden. Mit dem Steuerelement in der Standardversion wie mit obigem Code erzeugt können wir noch nicht allzu viel anfangen – außer den folgenden Aktionen:

- die Spaltenbreiten durch Ziehen der vertikalen Striche zwischen den Spalten,
- Ändern der Reihenfolge der Spalten durch ziehen einer Spalten per Drag and Drop an die gewünschten Stelle (siehe Bild 3) und
- Einstellen der optimalen Spaltenbreite durch einen Doppelklick auf den vertikalen Strich rechts von der anzupassenden Spalte.

Davon abgesehen ergeben sich keine Möglichkeiten ohne zusätzliche Programmierung. Praktischerweise passt das Steuerelement die Spaltenbreiten auch automatisch an die Inhalte an. Und, wie oben bereits erwähnt, können Sie im **ListView**-Element ohne weiteres Zutun mehrere Elemente gleichzeitig auswählen – im Falle von Bild 4 durch Anklicken der Einträge bei gedrückter **Strg**-Taste.

ListView: Sortierungen

Das ListView-Steuerelement bietet standardmäßig keine Möglichkeit, die angezeigten Einträge nach dem Inhalt zu sortieren. Wie Sie eine Funktion zum Sortieren der Daten hinzufügen, zeigen wir Ihnen anhand zweier Beispiele. Das erste nutzt einfache Schaltflächen, um die Daten nach dem Inhalt einer bestimmten Spalte zu sortieren. Die zweite sieht etwas professioneller aus und fügt jeder Spalte die Möglichkeit hinzu, die enthaltenen Daten per Mausklick auf den Spaltenkopf abwechselnd auf- oder absteigend zu sortieren.

Voraussetzung

Wir setzen auf dem Beispielprojekt an, das wir mi Artikel Das **ListView**-Steuerelement beschrieben haben. Hier haben wir ein **ListView**-Element mit den Daten einer Auflistung namens **Adressen** mit Elementen des Typs **Adresse** gefüllt.

Sortierung per Klick auf die Spaltenköpfe

Das Listenfeld im Windows Explorer liefert die Möglichkeit, per Klick auf die Spaltenköpfe in der Dateiliste die Sortierung anzupassen (siehe Bild 1). Dies wollen wir für das **ListView**-Element auch realisieren. Dazu nutzen wir eine neue Schaltfläche namens **btnSortByFirstName**, die wir wie folgt definieren:

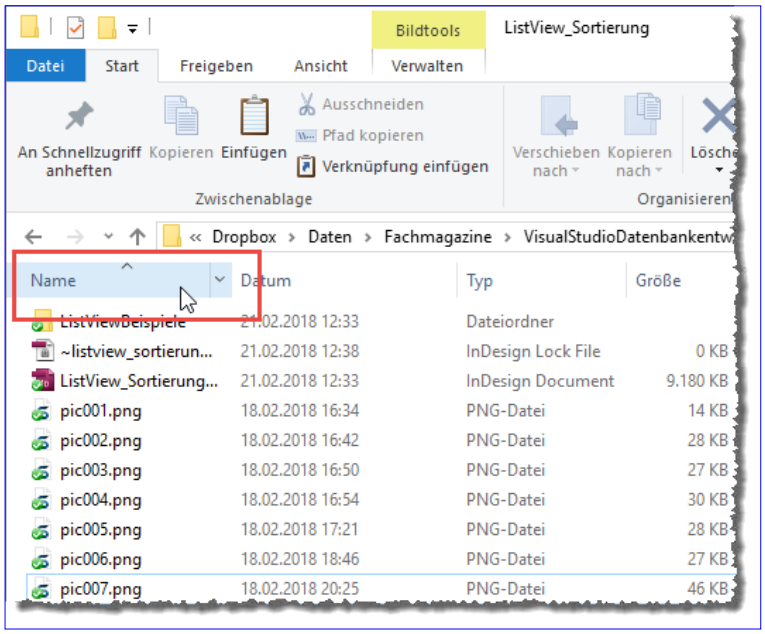


Bild 1: Sortierung im Windows Explorer

```
<Button x:Name="btnSortByFirstNameAsc" Click="btnSortByFirstNameAsc_Click">Aufsteigend nach Vorname sortieren</Button>
```

Und wir legen direkt noch eine Schaltfläche für die umgekehrte Sortierung an:

```
<Button x:Name="btnSortByFirstNameDesc" Click="btnSortByFirstNameDesc_Click">Absteigend nach Vorname sortieren</Button>
```

Für die Schaltflächen legen wir die folgenden beiden Ereignismethoden an:

```
Private Sub btnSortByFirstNameAsc_Click(sender As Object, e As RoutedEventArgs)
    Dim sortDescription As SortDescription
    sortDescription = New SortDescription("Vorname", ListSortDirection.Ascending)
    lvwAdressen.Items.SortDescriptions.Clear()
    lvwAdressen.Items.SortDescriptions.Add(sortDescription)
```

```
End Sub

Private Sub btnSortByFirstNameDesc_Click(sender As Object, e As RoutedEventArgs)
    Dim sortDescription As SortDescription
    sortDescription = New SortDescription("Vorname", ListSortDirection.Descending)
    lvwAdressen.Items.SortDescriptions.Clear()
    lvwAdressen.Items.SortDescriptions.Add(sortDescription)
End Sub
```

Diese legen ein neues Element des Typs **SortDescription** und übergeben diesem den Namen der zu sortierenden Spalte und die Sortierreihenfolge. Dann leeren sie die bisherigen Sortierungen in der Auflistung **SortDescriptions** einer weiteren Auflistung namens **Items** und fügen das neue Objekt über die Variable **sortDescription** zur Auflistung **SortDescriptions** hinzu. Als Ergebnis können Sie die Sortierreihenfolge der Spalte **Vorname** nun in beiden Richtungen festlegen (siehe Bild 2).

Mehrere Kriterien

Sie können nun auch mehrere Kriterien festlegen. Dazu erstellen Sie einfach immer weitere Objekte des Typs **SortDescription** und fügen diese der **SortDescriptions**-Auflistung hinzu. Wenn Sie möchten, dass die Daten erst nach dem Nachnamen und dann nach dem Vornamen aufsteigend sortiert werden, was bedeutet, dass erst nach dem Nachnamen sortiert wird und erst dann nach dem Vornamen, wenn es mehrere gleiche Nachnamen gibt, verwenden Sie etwa den folgenden Code:

```
Dim view As CollectionView
Dim sortDescription As SortDescription
view = CollectionViewSource.GetDefaultView(lvwAdressen.ItemsSource)
view.SortDescriptions.Clear()
sortDescription = New SortDescription("Nachname", ListSortDirection.Ascending)
view.SortDescriptions.Add(sortDescription)
```

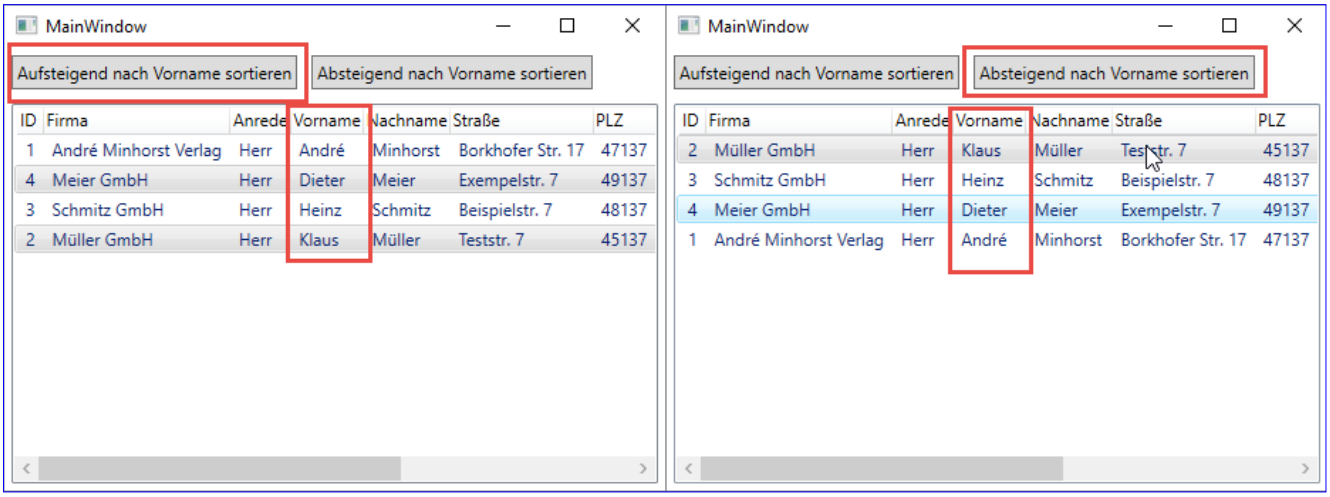


Bild 2: Zwei verschiedene Sortierungen

Visual Basic: Mit Klassen programmieren

Wer bisher mit VBA oder C# gearbeitet hat und zu VB wechseln möchte, sieht sich bei der Entwicklung von WPF-Anwendungen einigen Änderungen gegenüber. Dieser Artikel liefert die Grundlagen zum Umgang mit Namespaces und zur Programmierung von Klassen. Zum Experimentieren mit den Beispielen nutzen wir das Tool LINQPad 5.

Klassen

Im Gegensatz zu C# werden Klassen unter Visual Basic nicht durch eine geschweifte Klammer eingeleitet und beendet, sondern wir arbeiten mit dem für Visual Basis typischen Start- und Endzeilen:

```
Class Beispiel
    ---
End Class
```

Class-Elemente können dabei mit folgenden Modifizierern ausgestattet werden:

- **Public**: Die Klasse ist innerhalb der Assembly und auch von anderen Assemblies aus sichtbar.
- **Friend**: Die Klasse ist nur innerhalb der Assembly sichtbar, zu der sie gehört. Dies ist der Standard-Modifizierer, wenn Sie dem Schlüsselwort **Class** weder **Public** noch **Friend** oder einen anderen Modifizierer voranstellen!

Klasse erstellen

Eine Klasse wird typischerweise in einer eigenen Code-Datei angelegt. Wenn Sie ein neues WPF-Fenster anlegen, fügt Visual Studio beispielsweise eine eigene Datei mit der Dateiendung **.xaml.vb** hinzu, welche als Code behind-Modul für das WPF-Fenster dient. Wenn Sie eine neue Klasse benötigen, die nicht als Code behind-Modul für ein WPF-Element dient, erledigen Sie dies normalerweise über den Dialog **Neues Element hinzufügen...**, den Sie am einfachsten durch Markieren des Projekts im Projektmappen-Explorer und anschließendes Betätigen der Tastenkombination **Strg + Umschalt + A** aufrufen.

Hier wählen Sie dann den Eintrag **Klasse** aus und geben den gewünschten Namen an, in diesem Fall **Beispiel** (siehe Bild 1).

Die neue Klasse wird dann mit dem **Public**-Schlüsselwort ausgezeichnet. Wenn Sie nicht wollen, dass die Klasse in anderen Assemblies genutzt werden kann, ändern Sie **Public** also in **Friend**:

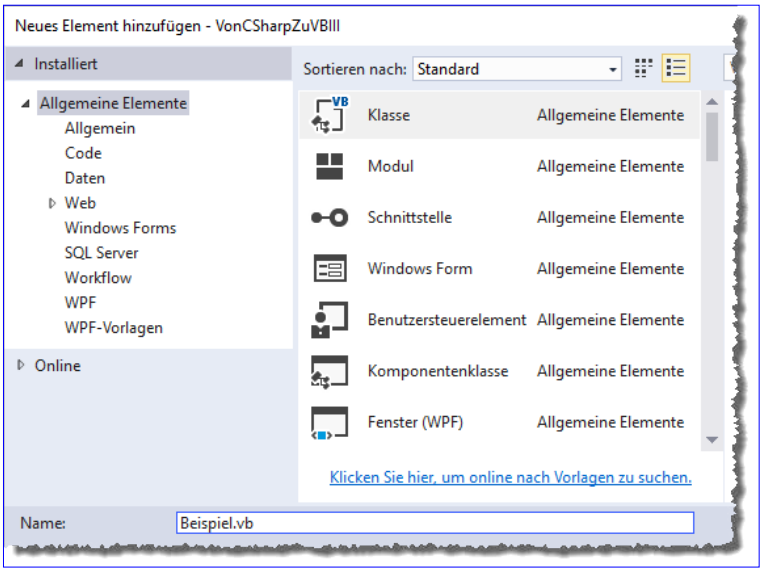


Bild 1: Einfügen einer neuen Klassendatei

```
Public Class Beispiel

End Class
```

Objekte auf Basis von Klassen initialisieren

Es gibt verschiedene Arten, eine Klasse zu instanzieren. Wenn Sie den Zeitpunkt der Deklaration und der Initialisierung trennen-möchten, nutzen Sie zwei Anweisungen:

```
Dim Beispielklasse As Beispiel
Beispielklasse = New Beispiel
```

Anderenfalls haben Sie die Wahl zwischen den folgenden beiden Schreibweisen. Die erste kennen Sie von VBA:

```
Dim Beispielklasse As New Beispiel
```

Die zweite ist eine Kombination aus dem Zweizeiler:

```
Dim Beispielklasse As Beispiel = New Beispiel
```

Weitere Klassen anlegen

Wenn Sie weitere Klassen anlegen wollen, haben Sie die Wahl: Sie können jeweils eine neue Datei mit jeder Klasse erstellen, was wie oben beschrieben funktioniert. Theoretisch können Sie aber auch mehrere Klassen in einer Datei speichern. Sie können neue Klassen auch innerhalb bestehender Klassen anlegen. Das sieht dann etwa wie folgt aus:

```
Public Class Beispiel
    Public Class UntergeordneteKlasse
    End Class
End Class
```

Mit Klassen experimentieren

Die schönste Benutzeroberfläche auch zum Experimentieren mit Klassen ist das Tool **LINQPad**. Damit können Sie, was für die Beispiele dieses Artikels sinnvoll ist, unter Language den Eintrag **VB Program** auswählen. Es wird dann automatisch eine **Sub Main** eingerichtet, unterhalb derer Sie die Klassen dieses Artikels definieren können. In **Sub Main** greifen Sie dann ganz einfach auf die Klassen zu (siehe Bild 2). Wenn Sie auf die **Start**-Schaltfläche klicken, wird **Sub Main** ausgeführt. Wir fügen der untergeordneten Klasse einfach eine öffentliche Eigenschaft hinzu:

```
Public Class Beispiel
    Public Class UntergeordneteKlasse
        Public Property Test As String = "bla"
    End Class
End Class
```

Die untergeordnete Klasse können Sie etwa wie folgt deklarieren und initialisieren:

```
Dim UntergeordneteKlasse As Beispiel.UntergeordneteKlasse
UntergeordneteKlasse = New Beispiel.UntergeordneteKlasse
Debug.WriteLine (UntergeordneteKlasse.Test)
```

Für solche untergeordneten Klassen gibt es neben den Modifizierern **Public** und **Friend** noch drei weitere mögliche Modifizierer. Diese Modifizierer gelten natürlich auch für die anderen Elemente einer Klasse, also beispielsweise Methoden, Eigenschaften oder Ereignisse:

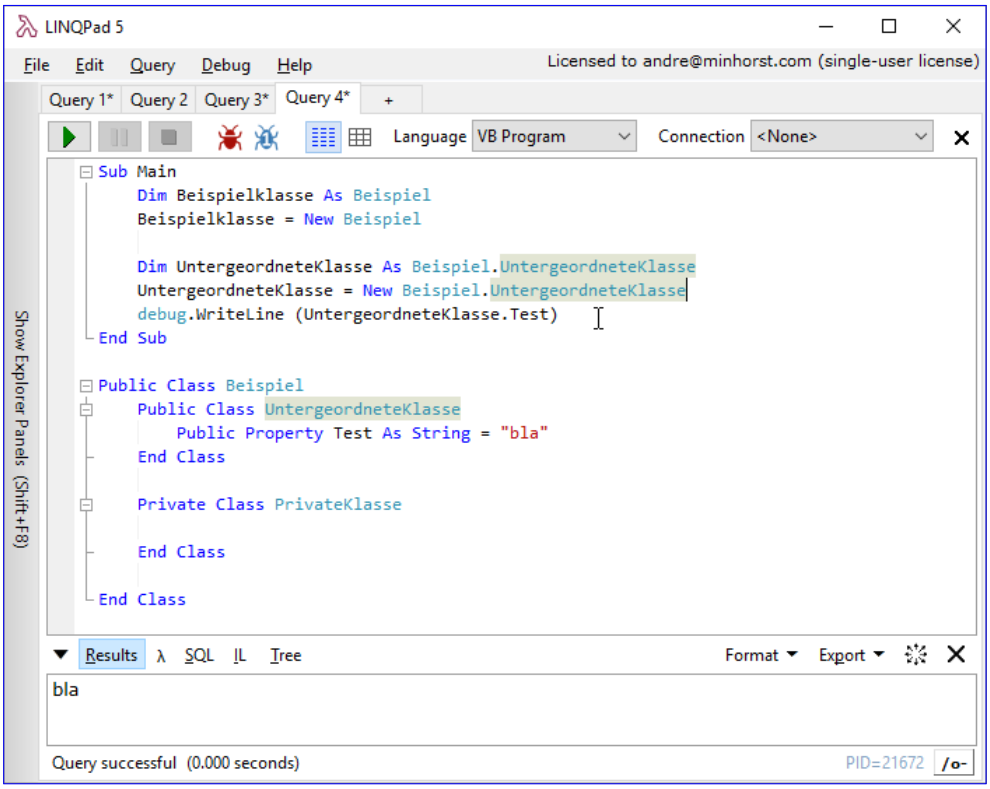


Bild 2: Experimentieren mit Klassen in LINQPad

- **Private**: Die Klassen, Methoden, Ereignisse und Eigenschaften können nur innerhalb des Objekts genutzt werden, in der diese auch definiert wurde. Außerhalb der übergeordneten Klasse ist sie nicht sichtbar. Wenn Sie die Klasse **UntergeordneteKlasse** von oben als **Private** deklarieren würden, könnten Sie nicht von der Methode **Public Main** darauf zugreifen.

- **Protected**: Die Klasse, Methoden, Ereignisse und Eigenschaften können nur innerhalb des Objekts genutzt werden, indem diese definiert wurden, und in Objekten auf Basis von Klassen, die von der ursprünglichen Klasse abgeleitet sind. Ableitungen von Klassen stellen wir später noch vor.

- **Protected Friend**: Die Klasse, Methoden, Ereignisse und Eigenschaften können nur innerhalb des Objekts und der Assembly genutzt werden, indem diese definiert wurden, und in Objekten auf Basis von Klassen, die von der ursprünglichen Klasse abgeleitet sind.

Felder von Klassen

Sie können einer Klasse sogenannte Felder hinzufügen, indem Sie direkt unterhalb der **Class**-Ebene Variablen deklarieren. Dies können Variablen mit dem Modifizierer **Public**, **Private** oder einem der anderen Modifizierer sein. **Public**-Felder können Sie

auch von außerhalb lesen und schreiben. Die Klasse im folgenden Beispiel hat eine öffentliche und eine private Variable. Die öffentliche ist von außen per IntelliSense verfügbar und kann sowohl geschrieben als auch abgefragt werden:

```
Sub Main
    Dim KlasseMitFeld As KlasseMitFeld
    KlasseMitFeld = New KlasseMitFeld
    KlasseMitFeld.Zeichenkette = "Test"
    Debug.WriteLine(KlasseMitFeld.Zeichenkette)
End Sub

Public Class KlasseMitFeld
    Public Zeichenkette As String
    Private Zahl As Integer
End Class
```

Felder als nur lesbar markieren

Wenn ein Feld nur lesenden, aber keinen schreibenden Zugriff ermöglichen soll, deklarieren Sie es mit dem Schlüsselwort **ReadOnly**:

```
Public ReadOnly ZeichenketteNurLesbar As String
```

Diese liefert dann beim Zugriff von außen den Wert **null** zurück. Schreibend können Sie von außen nicht auf dieses Feld zugreifen, und von innerhalb der Klasse, in der es definiert ist, auch nur sehr eingeschränkt. Sie können es in der Deklarationszeile mit dem gewünschten Wert initialisieren:

```
Public ReadOnly ZeichenketteNurLesbar As String = "Nur lesbar"
```

Die Konstruktor-Methode

Die einzige weitere Alternative bei schreibgeschützten Feldern ist die Verwendung der Konstruktor-Methode. Also schauen wir uns diese gleich hier an. Die Konstruktor-Methode ist wie unter C# die Methode, die beim Initialisieren eines Objekts auf Basis der Klasse automatisch ausgeführt wird.

Unter C# heißt diese wie die Klasse selbst, unter Visual Basic ist dies ganz anders: Hier lautet der Name der Konstruktor-Methode schlicht **New**. Die Konstruktor-Methode der folgenden Klasse **KlasseMitFeld** weist dem Feld **ZeichenketteNurLesbar** eine Zeichenkette zu. Die Methode **Sub Main** initialisiert die Klasse und gibt dann als Beweis die zugewiesene Zeichenkette aus:

```
Sub Main
    Dim KlasseMitFeld As New KlasseMitFeld
    Debug.WriteLine(KlasseMitFeld.ZeichenketteNurLesbar)
End Sub
```

```
Public Class KlasseMitFeld
    Public ReadOnly ZeichenketteNurLesbar As String
    Public Sub New
        ZeichenketteNurLesbar = "Wert von der Konstruktor-Methode zugewiesen"
    End Sub
End Class
```

Die Konstruktor-Methode ist ein von vielen VBA-Entwicklern schmerzlich vermisstes Feature: Damit können Sie einer Klasse direkt beim Instanzieren wichtige Informationen übermitteln, die dann auch direkt verarbeitet werden können – beispielsweise, um einer **Kunde**-Klasse direkt den Namen und die Adresse des Kunden zu übermitteln.

Eigenschaften

Felder können fast wie Eigenschaften genutzt werden – es gibt schreib- und lesbar Felder oder auch nur lesbare Felder (**Read-Only**). Eigenschaften erlauben aber eine genauere Steuerung der Berechtigungen und erlauben auch noch das Ausführen weiterer Aktionen, wenn der Eigenschaftswert geschrieben oder gelesen wird. Eine Eigenschaft enthält in der Regel eine lokale Variable, die den Eigenschaftswert speichert (die sogenannte Member-Variable), sowie ein öffentliches **Property**-Konstrukt, welches jeweils einen sogenannten Getter und einen Setter enthält. Das sieht dann insgesamt wie folgt aus:

```
Public Class KlasseMitEigenschaften
    Private _eigenschaft As String
    Public Property Eigenschaft As String
        Get
            Return _eigenschaft
        End Get
        Set(ByVal value As String)
           _eigenschaft = value
        End Set
    End Property
End Class
```

Damit sieht die Notation erheblich anders aus als unter C#, und auch wer von VBA kommt, muss sich etwas umorientieren. Unter VBA gab es noch jeweils eine **Property Get**- und eine **Property Set**-Eigenschaft, wobei die **Property Get**-Eigenschaft den Rückgabewert an eine Variable mit dem Namen der Eigenschaft übergeben hat. Der schreibende und lesende Zugriff auf diese Eigenschaft erfolgt dann etwa wie folgt:

```
Dim klasseMitEigenschaften = New KlasseMitEigenschaften
klasseMitEigenschaften.Eigenschaft = "Test"
Debug.WriteLine(klasseMitEigenschaften.Eigenschaft)
```

Wenn Sie eine einfache schreib- und lesbare Eigenschaft wie oben definieren wollen, können Sie das übrigens auch per Einzeiler erledigen – als sogenannte Auto-Implemented Property:

```
Public Property Eigenschaft As String
```

Genau genommen ist das die schnellste und übersichtlichste Art, eine einfache Property zu definieren. Sie brauchen die andere Schreibweise im Grunde nur, wenn Sie entweder für den Setter oder den Getter zusätzlich auszuführende Anweisungen benötigen, die beim Setzen oder Lesen der Eigenschaft benötigt werden. Oder wenn Sie die Property mit eingeschränkten Zugriffsberechtigungen versehen wollen – dazu später mehr.

Selbst, wenn Sie innerhalb der Klasse auf den Inhalt der privaten Variablen zugreifen wollen, welche den der Property zugewiesenen Wert enthält, brauchen Sie nur die einfache Variante zu nutzen. Im folgenden Beispiel definieren wir eine Klasse mit einer Property namens **Test** als Auto-Implemented Property.

Eine zweite Property namens **Test2** soll den Wert der Variablen **_Test** zurückliefern. **_Test** ist normalerweise die private Membervariable zum Speichern des Wertes der Property **Test**. Sie ist hier nicht explizit deklariert. Dennoch können Sie über diese Variable nach dem Zuweisen eines Wertes den Wert der Property **Test** lesen und beispielsweise über die Property **Test2** zurückgeben:

```
Sub Main
    Dim klasseMitProperty As New KlasseMitProperty
    klasseMitProperty.Test = "bla"
    Debug.WriteLine(klasseMitProperty.Test2)
End Sub

Public Class KlasseMitProperty
Public Property Test As String
    Public Readonly Property Test2 As String
        Get
            Return _Test
        End Get
    End Property
End Class
```

Property mit Getter und Setter schnell anlegen

Visual Studio hat eine tolle Vereinfachung für das Anlegen von Properties mit **Get**- und **Set**-Element. Dabei gehen Sie so vor:

- Schreiben Sie die Property in der kurzen Schreibweise, also also Auto-Implemented Property, in die erste Zeile:
  Public Property Test As String

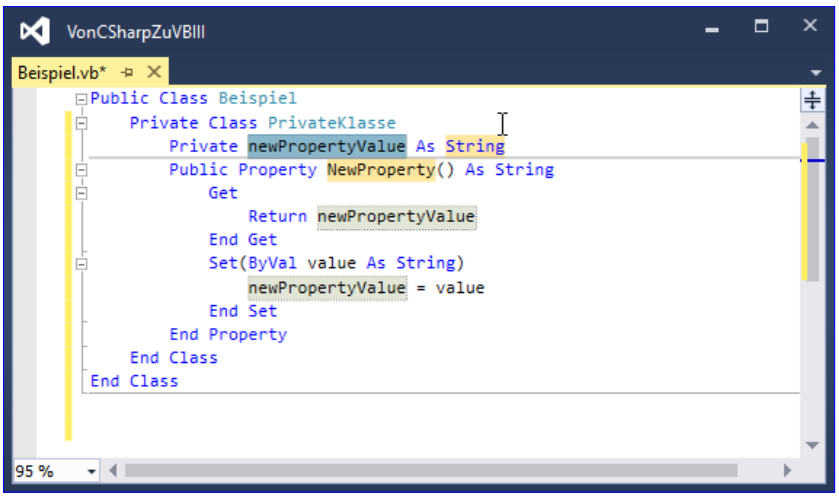


Bild 3: Vorlage für eine Property

EDM: Der Code First-Ansatz

In den bisherigen Ausgaben haben wir immer mit dem Database First-Ansatz gearbeitet, das heißt, dass wir unser Entity Data Model auf Basis einer bestehenden Datenbank im SQL Server oder SQLite generiert haben. Das geht auch andersherum: Sie erstellen ein paar Klassen, geben eine Verbindungszeichenfolge an und beim ersten Versuch, auf die Daten der Datenbank zuzugreifen, erstellt die Anwendung automatisch die Datenbank im angegebenen SQL Server. Wie das gelingt und wie die Klassen und die Verbindungszeichenfolge aussehen müssen, erfahren Sie in diesem Beitrag.

Als Beispiel wählen wir unsere Anwendung zum Erstellen von Anschreiben mit Word von einem WPF-Fenster aus. Dieser haben wir im ersten Schritt ja nur einfache Textfelder als Eingabemöglichkeit hinzugefügt, in die der Benutzer Anschrift, Datum, Betreff und Inhalt des Anschreibens eintragen kann, bevor daraus ein Word-Dokument auf Basis der gewählten Vorlage erstellt wird. Nun wollen wir dies erweitern – und zwar um die Möglichkeit, sowohl verschiedene Anschriften zu verwalten als auch um die Anschreiben abzuspeichern, um diese gegebenenfalls später noch einmal aufzurufen oder auch als Vorlage für neue Anschreiben zu verwenden.

Unter Access würden wir das Formular einfach an eine Tabelle zum Speichern der Daten des Anschreibens binden und gegebenenfalls noch eine Tabelle für die verschiedenen Anschriften hinzufügen. Allerdings wollen wir ja in diesem Magazin lernen, mit den unter WPF gängigen Techniken umzugehen – wie etwa dem Entity Data Model als Möglichkeit, Daten zwischen Benutzeroberfläche und Datenbank hin- und herzuschieben.

In den vorherigen Ausgaben haben wir meist eine vorhandene Datenbank genutzt und mit dem dafür vorgesehenen Assistenten ein Entity Data Model erstellen lassen. Dieses enthält die Klassen und Auflistungen, mit denen wir dann von der Benutzeroberfläche beziehungsweise von der Anwendungslogik aus auf die Daten der Datenbank zugreifen konnten. In diesem Beispiel wollen wir es einmal andersherum versuchen: Wir wollen zuerst die Klassen für die beiden Entitäten programmieren, also für die Anschreiben und die Adressen, und daraus dann ein Datenmodell generieren. Auch dieser Fall ist vom Entity Framework als Möglichkeit vorgesehen und nennt sich Code First. Wir erstellen also erst den Code und lassen daraus dann die Datenbank erzeugen. Das

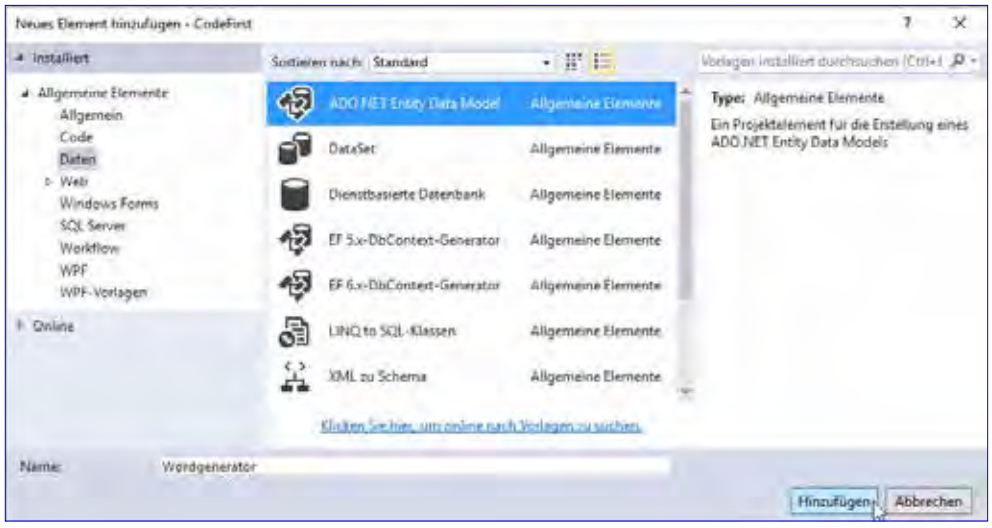


Bild 1: Entity Data Model zum Projekt hinzufügen

wir für das erste Beispiel eine relativ einfache Konstellation mit nur zwei Entitäten nutzen (okay, mit Anreden für die Adressen vielleicht drei), wird die Komplexität ein wenig vermindern.

Vorbereitungen

Als Erstes erstellen wir ein neues, leeres Projekt namens **CodeFirst** – und zwar mit der Vorlage **Visual Basic\Windows\WPF-Anwendung** oder **Visual C#\Windows\WPF-Anwendung**. Diesem Projekt fügen Sie nach Markierung des Eintrags mit dem Projektnamen **CodeFirst** im Projektmappen-Explorer ein Element des Typs **ADO.NET Entity Data Model** hinzu. Den dazu nötigen Dialog **Neues Element hinzufügen...** öffnen Sie mit der Tastenkombination **Strg + Umschalt + A**. Hier wählen Sie den passenden Elementtyp aus und legen als Name den Wert **Wordgenerator** fest (siehe Bild 1).

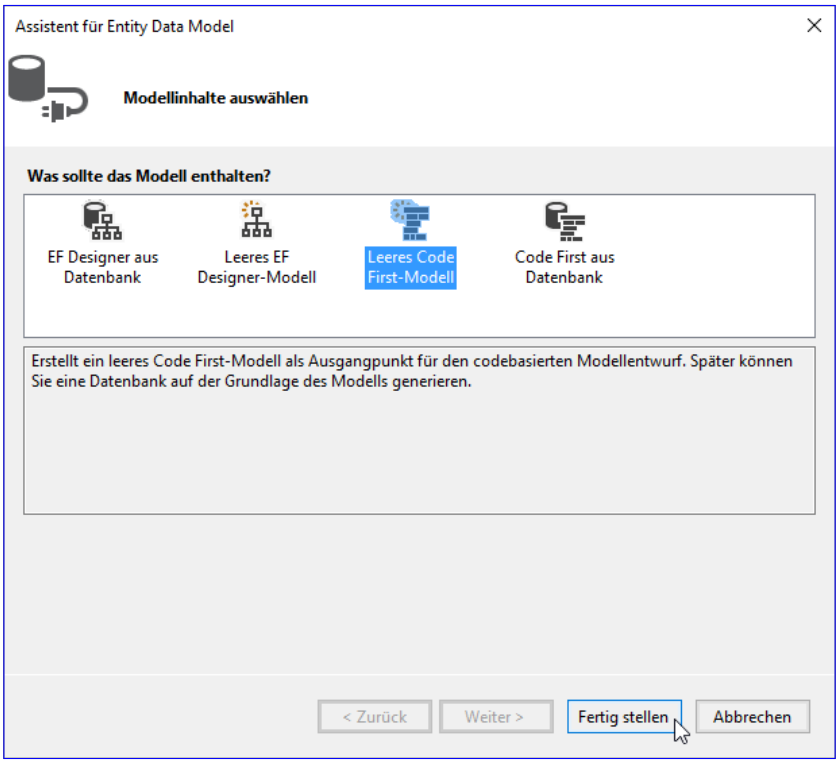


Bild 2: Modelltyp **Leeres Code First-Modell** auswählen

Im folgenden **Assistent für Entity Data Model** wählen Sie den Eintrag **Leeres Code First-Modell** aus (siehe Bild 2). Im Gegensatz zu der Variante, die wir sonst gewählt haben, nämlich **EF Designer aus Datenbank**, ist dies gleichzeitig schon der letzte Schritt in diesem Assistenten, den wir mit einem Klick auf die Schaltfläche **Fertigstellen** beenden.

Diesmal geschieht auch gar nicht viel in unserem Projekt. Die augenscheinlichste Änderung ist, dass der Assistent ein Klassenmodul namens **Wordgenerator.vb** (beziehungsweise **Wordgenerator.cs** unter C#) angelegt hat (siehe Bild 3). Die Klasse Wordgenerator erbt von der Klasse **DbContext**.

Modell programmieren

Nun programmieren wir die Klassen für unser Modell, auf dessen Basis wir später eine Datenbank erstellen lassen wollen.

Dazu legen wir drei neue Klassen an – jeweils wieder über die Tastenkombination **Strg + Umschalt + A** und den dadurch aufgerufenen Dialog **Neues Element einfügen...**. Hier wählen wir nun jeweils den Eintrag Klasse aus und legen die Namen **Anschreiben.vb**, **Adresse.vb** und **Anrede.vb** fest. Die Klasse **Anrede.vb** sieht wie folgt aus:

```
Public Class Anrede
    Public Property ID As Integer
    Public Property Anrede As String
```

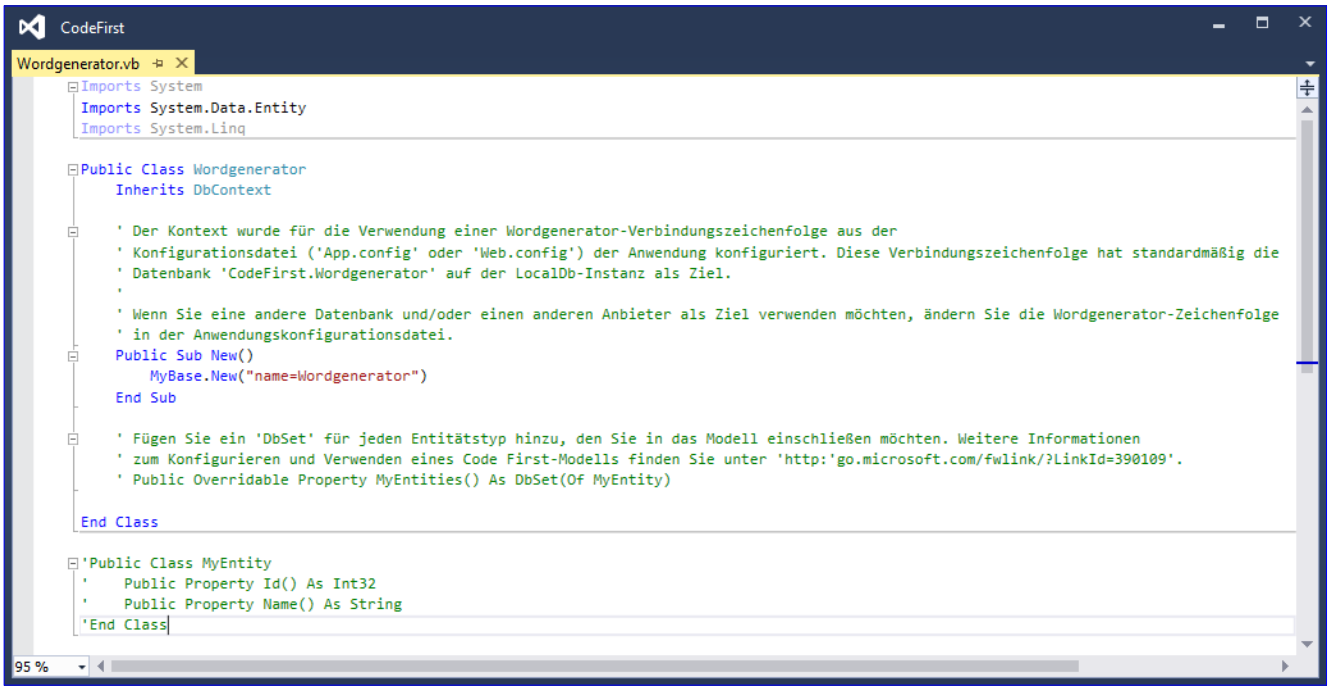


Bild 3: Durch den Assistenten angelegte Klasse

```
End Class
```

Für die Klasse Adresse legen wir den folgenden Code fest, wobei wir als Typ der Eigenschaft Anrede auch gleich die frisch angelegte Klasse gleichen Namens verwenden:

```
Public Class Adresse
    Public Property ID As Integer
    Public Property Firma As String
    Public Property Anrede As Anrede
    Public Property Vorname As String
    Public Property Nachname As String
    Public Property Strasse As String
    Public Property PLZ As String
    Public Property Ort As String
    Public Property Land As String
    Public Property EMail As String
End Class
```

Nachdem wir die Klasse **Adresse** erzeugt haben, die wir auch als Collection der Klasse **Anrede** zuordnen wollen – jede Anrede gehört ja zu keiner, einer oder mehreren Adressen –, können wir die Klasse **Anrede** noch um die Auflistung **Adressen** mit dem Typ **Adresse** erweitern:

```
Public Class Anrede
```

```
    ...
    Public Property Adressen As ICollection(Of Adresse)
End Class
```

Die Klasse **Anschreiben** erhält schließlich die folgenden Eigenschaften:

```
Public Class Anschreiben
    Public Property ID As Integer
    Public Property Adresse As Adresse
    Public Property Datum As Date
    Public Property Betreff As String
    Public Property Inhalt As String
End Class
```

Auch hier können wir nun die Klasse **Adresse** um die Auflistung der Anschreiben ergänzen:

```
Public Class Adresse
    ...
    Public Property Anschreiben As ICollection(Of Anschreiben)
End Class
```

Mit der Angabe der beiden **ICollections** in den Klassen **Anschreiben** und **Adresse** legen wir praktisch die 1:n-Beziehung zwischen den Klassen fest. Damit kann dann eine Adresse mehreren Anschreiben zugeordnet werden und eine Anrede mehreren Adressen.

Auflistungen in der DBContext-Klasse definieren

Nun gehen wir zu der eingangs erwähnten Klasse namens Wordgenerator zurück, die ja vom Assistenten automatisch angelegt wurde und die von der Klasse **DBContext** erbt. Hier legen wir nun noch Auflistungen eines speziellen Typs, nämlich **DBSet**, für die drei Entitäten **Anrede**, **Adresse** und **Anschreiben** fest. Das sieht für die ersten beiden Elemente wie folgt aus:

```
Public Property Anreden() As DbSet(Of Anrede)
Public Property Adressen() As DbSet(Of Adresse)
```

Beim Versuch, das dritte Element für die Anschreiben einzutragen, stellen wir fest, dass wir mit **Anschreiben** eine schlechte Bezeichnung gewählt habe – diese ist nämlich für Plural und Singular gleich, was eine Unterscheidung unmöglich macht. Also ändern wir den Namen der Klasse **Anschreiben** noch in **Brief**. Am einfachsten geht dies, indem Sie die Klasse schnell neu unter dem gewünschten Namen anlegen und den Inhalt in die neue Klasse kopieren:

```
Public Class Brief
    ...
End Class
```

Die Eigenschaft **Anschreiben** in der Klasse **Adresse** müssen wir dementsprechend wie folgt anpassen:

```
Public Property Briefe As ICollection(Of Brief)
```

Die alte Klasse **Anschreiben** können wir dann aus dem Projekt löschen. Außerdem tragen wir noch die fehlende Zeile für die Briefe in der Klasse **Wordgenerator** nach:

```
Public Property Briefe() As DbSet(Of Brief)
```

Damit haben wir die grundlegende Definition der Klassen abgeschlossen und schauen uns an, wie wir damit eine Datenbank erstellen können.

Verbindungszeichenfolge festlegen

Dazu benötigen wir als Erstes eine geeignete Verbindungszeichenfolge. Diese hat der Assistent vermutlich bereits für uns angelegt. Ob das der Fall ist, erfahren wir, indem wir uns die Datei **App.Config** des Projects ansehen. Diese sieht in unserem Fall wie in Bild 4 aus. Hier finden wir auch bereits einen Eintrag namens **<connectionStrings>** mit einem Untereintrag **add**. Dieser enthält die Eigenschaft connectionString mit der Verbindungszeichenfolge. Diese gibt aktuell als Server **(LocalDb)\MSSQLLocalDB** an und nennt die Datenbank **CodeFirst.Wordgenerator**. Diesen Namen ändern wir in **WordGenerator**. Die Bezeichnung der Verbindungszeichenfolge aus dem Attribut **Name** lautet **Wordgenerator** und wird auf Basis der Bezeichnung der Klasse benannt, die von **DbContext** erbt. Diese Einstellung wollen wir vorerst beibehalten. **LocalDb** ist die SQL Server-Datenbank, die mit Visual Studio installiert wird und während der Entwicklung die einfachste Möglichkeit für den Zugriff auf Datenbanken bietet. Die Verbindungszeichenfolge lautet nun:

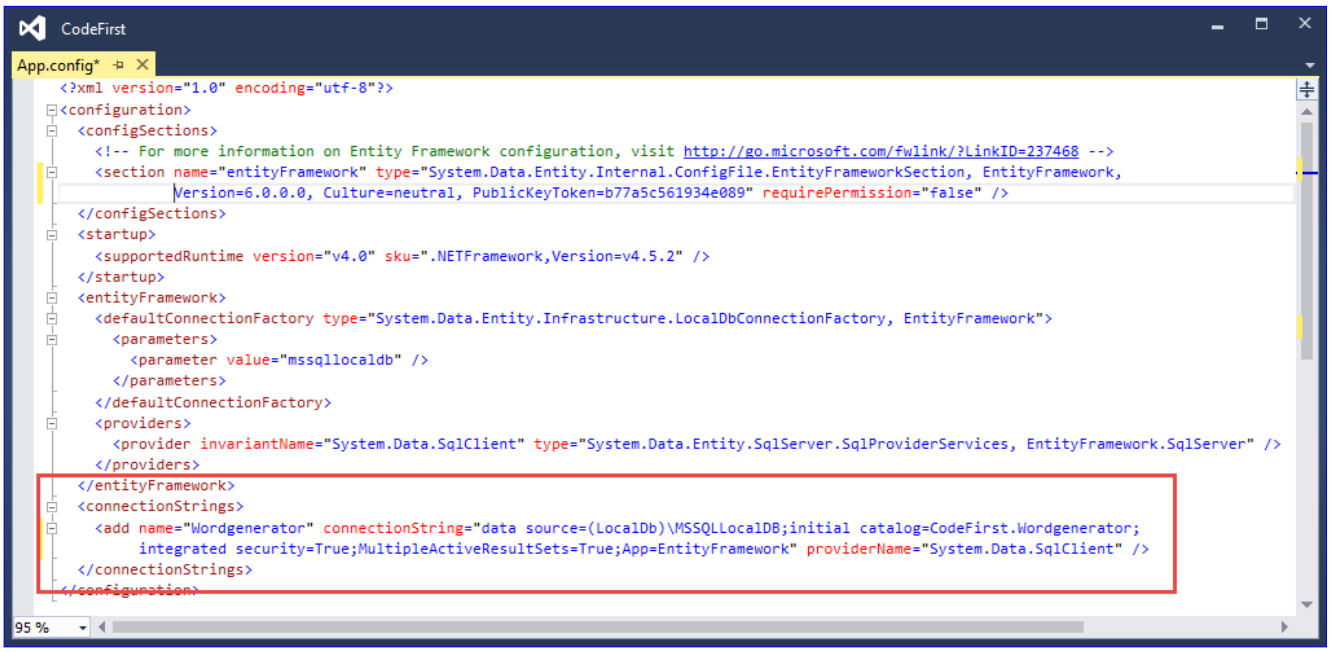


Bild 4: Die vom Assistenten automatisch angelegte Verbindungszeichenfolge

EDM: Code First - Datenbank erweitern

Im Artikel »EDM: Der Code First-Ansatz« haben wir uns angesehen, wie Sie auf Basis eines frisch erstellten Entity Data Models eine Datenbank samt den nötigen Tabellen erstellen können – und zwar automatisch beim ersten Zugriff auf die noch nicht vorhandene Datenbank. Nun gehen wir einen Schritt weiter und zeigen, wie Sie auch noch Änderungen am Entity Data Model auf bestehende Datenbanken mit älterem Versionsstand übertragen und somit ein kombiniertes Update von Anwendung und Datenbank ausliefern können.

Aktualisierungen am bestehenden Datenmodell

Wir haben während der Entwicklung der Anwendung, wir wir es im Artikel **EDM: Der Code First-Ansatz** gelernt haben, die Freiheit, nach Lust und Laune die Zieldatenbank zu löschen und neu zu erstellen. Wohlgemerkt: während der Entwicklung! Sobald die Datenbank jedoch einmal mit Daten gefüllt ist oder sogar beim Kunden läuft, wollen Sie Aktualisierungen am Datenmodell sicher etwas eleganter und ohne Datenverlust übermitteln. Auch dazu bietet Code First Möglichkeiten.

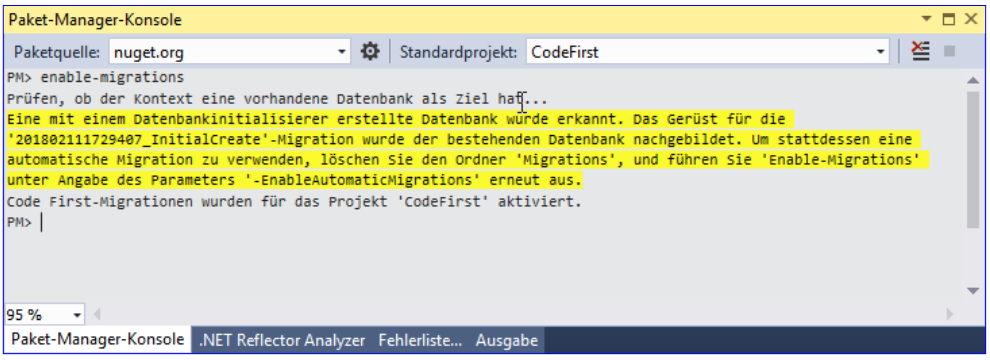


Bild 8: Aktivieren der Migrationsfunktion

Um Migrationen mit Code First durchzuführen, also die Änderungen erst am bestehenden Modell der Entitäten durchzuführen und diese dann auf die Tabellen der Datenbank zu übertragen, müssen wir zunächst die Migrationsfunktion aktivieren.

Dazu benötigen wir die Paket-Manager-Konsole, die Sie mit dem Menübefehl **Ansicht|Weitere Fenster|Paket-Manager-Konsole** einblenden. Ist dieser eingeblendet, wählen Sie oben rechts unter Standardprojekte den Namen des betroffenen Projekts aus, in diesem Fall **CodeFirst**. Dann geben Sie den folgenden Befehl ein:

```
PK> enable-migrations
```

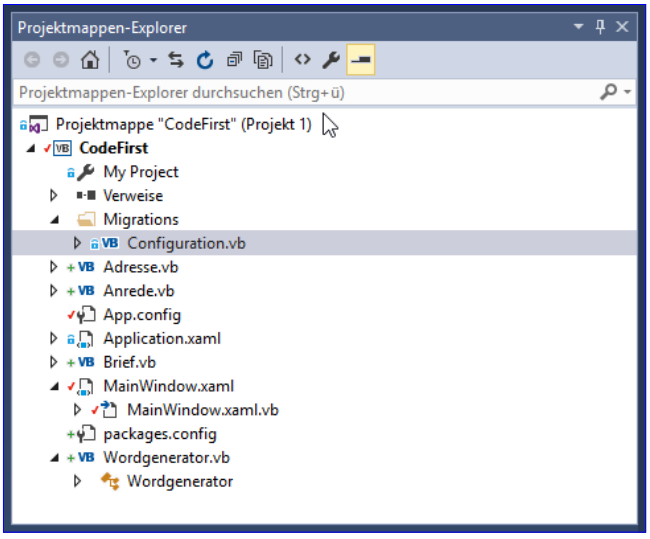


Bild 9: Aktivieren der Migrationsfunktion

Das Ergebnis finden Sie in Bild 1 vor.

Im Projekt haben sich nun einige Änderungen ergeben, die Sie direkt im Projektmappen-Explorer ablesen können. Dort finden Sie nun nämlich einen neuen Ordner namens **Migrations** (siehe Bild 2). Dieser enthält eine Datei namens **Configuration.vb**.

Diese Datei enthält vor allem die **Seed**-Methode. Dieser können Sie Anweisungen zum Schreiben von Daten in die Tabellen der Datenbank hinzufügen. Die **Seed**-Methode wird später beim Update des Datenbank-Backends ausgelöst. Deshalb fügen wir beispielsweise die folgenden Codezeilen zu dieser Methode in der Klasse **Configuration.vb** hinzu:

```
Namespace Migrations
    Friend NotInheritable Class Configuration
        Inherits DbMigrationsConfiguration(Of Wordgenerator)
        Public Sub New()
            AutomaticMigrationsEnabled = False
            ContextKey = "CodeFirst.Wordgenerator"
        End Sub
        Protected Overrides Sub Seed(context As Wordgenerator)
            Dim anrede As New Anrede With {.Bezeichnung = "Herr"}
            context.Anreden.AddOrUpdate(Function(p) p.Bezeichnung, anrede)
            anrede = New Anrede With {.Bezeichnung = "Frau"}
            context.Anreden.AddOrUpdate(Function(p) p.Bezeichnung, anrede)
            anrede = New Anrede With {.Bezeichnung = "Firma"}
            context.Anreden.AddOrUpdate(Function(p) p.Bezeichnung, anrede)
        End Sub
    End Class
End Namespace
```

Damit wollen wir sicherstellen, dass zumindest die grundlegenden Daten der Anwendung, nämlich die Anreden, bereits in die Tabelle **Anreden** geschrieben werden – hier also die Datensätze mit den Werten **Herr**, **Frau** und **Firma** im Feld **Bezeichnung**.

Anschließend geben Sie einen weiteren Befehl in den Bereich **Paket-Manager-Konsole** ein (siehe Bild 3):

```
PM> Add-Migration Initial
```

Dadurch wird eine weitere Datei zum Ordner **Migrations** hinzugefügt, diesmal nach dem Schema **201802121707380_Initial.vb**. Diese Klasse enthält eine Methode namens **Up**, die

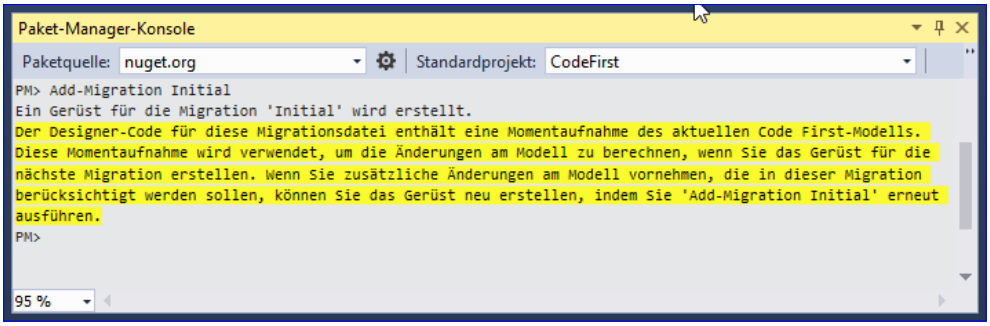


Bild 1: Initiale Migration

drei **CreateTable**-Anweisungen enthält, mit denen die Tabellen entsprechend der Entitäten **Anrede**, **Adresse** und **Brief** erstellt werden – hier in gekürzter Form nur mit den Anweisungen für die Tabelle **Anreden**:

```
Public Overrides Sub Up()
    ...
    CreateTable(
        "dbo.Anreden",
        Function(c) New With
            {
                .ID = c.Int(nullable := False, identity := True),
                .Bezeichnung = c.String()
            }) _
        .PrimaryKey(Function(t) t.ID)
    ...
End Sub
```

Zusätzlich enthält die Klasse noch eine Methode namens **Down**, welche die bestehenden Schlüssel, Indizes und Tabellen löscht:

```
Public Overrides Sub Down()
    DropForeignKey("dbo.Briefe", "Adresse_ID", "dbo.Adressen")
    DropForeignKey("dbo.Adressen", "Anrede_ID", "dbo.Anreden")
    DropIndex("dbo.Briefe", New String() { "Adresse_ID" })
    DropIndex("dbo.Adressen", New String() { "Anrede_ID" })
    DropTable("dbo.Briefe")
    DropTable("dbo.Anreden")
    DropTable("dbo.Adressen")
End Sub
```

Außerdem setzen Sie im gleichen Bereich noch den folgenden Befehl ab:

```
PM> Update-Database
```

Damit rufen Sie zunächst die Methoden der Klasse **201802121707380_Initial.vb** auf, welche die Datenbank und das Datenmodell anlegt, wenn diese noch nicht vorhanden ist, und dann mit der **Seed**-Methode die initialen Daten in die Tabelle **Anreden** schreibt.

Änderungen am Datenmodell

Nun nehmen wir eine kleine Änderung am Datenmodell vor. Wohlgemerkt: Die aktuelle Version, die durch die Datei **201802121707380_Initial.vb** beschrieben wird, haben wir durch die **Update-Database**-Methode in der **Paket-Manager-Konsole** bereits angewendet und damit eine neue Datenbank erzeugt. Nun fügen wir der Entität **Anreden** eine Eigenschaft namens **Briefanrede** hinzu:

```
<Table("Anreden")>
Public Class Anrede
    Public Property ID As Integer
    Public Property Bezeichnung As String
    Public Property Briefanrede As String
    Public Property Adressen As ICollection(Of Adresse)
End Class
```

Nun rufen wir wieder eine neue Anweisung in der **Paket-Manager-Konsole** auf:

```
add-migration add_Anrede_Briefanrede
```

Dies erstellt eine neue VB-Klasse namens **201802121730445_add_Anrede_Briefanrede.vb**, welche die Änderungen im Datenmodell seit der letzten Änderung enthält, und die wieder in Form der beiden Methoden **Up** und **Down** eingetragen werden. Das sieht dann so aus:

```
Public Partial Class add_Anrede_Briefanrede
    Inherits DbMigration
    Public Overrides Sub Up()
        AddColumn("dbo.Anreden", "Briefanrede", Function(c) c.String())
    End Sub
    Public Overrides Sub Down()
        DropColumn("dbo.Anreden", "Briefanrede")
    End Sub
End Class
```

Die **Up**-Methode wird ausgeführt, wenn die Migration angewendet werden soll, die **Down**-Methode, wenn diese rückgängig gemacht werden soll, also eine vorheriger Stand wiederhergestellt werden soll (mehr dazu weiter unten).

Damit rufen wir nun wieder den Befehl **update-database** in der **Paket-Manager-Konsole** auf, aktualisieren die im SQL Server-Objekt-Explorer angezeigten Datenbanken und navigieren wieder zu unserer Datenbank **Wordgenerator**.

Dort überzeugen uns davon, dass das neue Feld **Briefanrede** auch in der Tabelle **Anreden** vorhanden ist – was der Fall ist (siehe Bild 4).

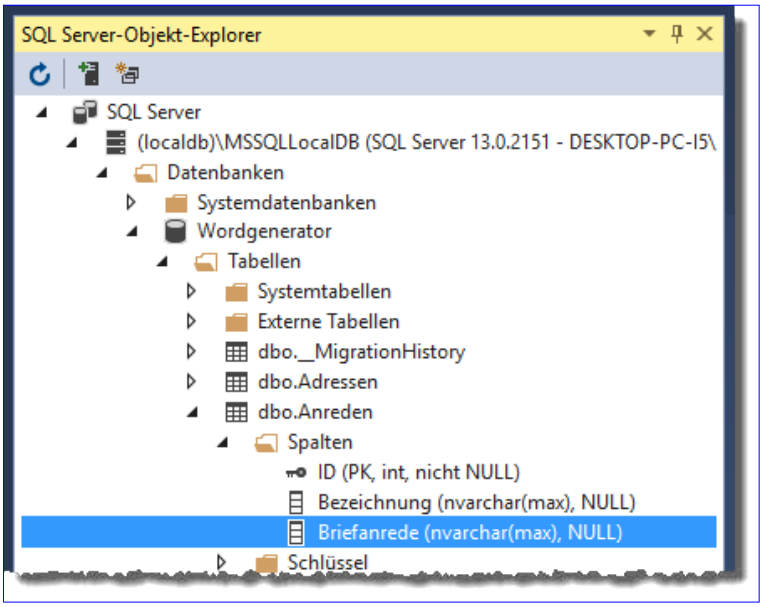


Bild 2: Das neue Feld ist in der Datenbank angekommen.

Datenbank komplett neu erstellen

Wir machen noch einen Test und löschen die Datenbank auf dem SQL Server-Objekt-Explorer, um diese mit der Anweidung **update-database** erneut zu erstellen (vor dem Löschen müssen Sie alle gegebenenfalls in Visual Studio geöffneten Datenbank-Elemente schließen). Übrigens können Sie, wenn Sie das Flag **-verbose** hinter **update-database** angeben, auch die ausgeführten SQL-Anweisungen in der **Paket-Manager-Konsole** ausgeben lassen.

Nach dem Vorgang, für den die **Down**-und **Up**-Methoden aller bisher im Verzeichnis Migrations gespeicherten **Migrations**-Klassen sowie die **Seed**-Methode der **Configuration.vb**-Klasse ausgeführt werden, finden wir alle gewünschten Elemente in der neuen Datenbank vor. Das heißt also, dass wir mit **update-database** sowohl die komplette Datenbank erstellen als auch zwischenzeitlich durchgeführte und in Form einer entsprechenden VB-Klasse gespeicherte Änderungen auf vorherige Versionen der Datenbank anwenden können.

Bisherige Operationen ermitteln

Wenn Sie wissen wollen, welchen Stand die aktuell verbundene Datenbank hat, setzen Sie in der Paket-Manager-Konsole den Befehl **get-migrations** ab. Dies liefert etwa die folgende Ausgabe:

```
PM> get-migrations
Auf die Zieldatenbank angewendete Migrationen werden abgerufen.
201802121737301_add_Anrede_Briefanrede
201802121736065_Initial
```

Downgrade

Sie können auch zu einer bestimmten Version zurückkehren. Dazu geben Sie den Befehl **update-database** mit dem Parameter **TargetMigration**, dem Doppelpunkt und den Namen der Migration an, also etwa so:

```
PM> update-database -TargetMigration:Initial
```

Damit wird der Stand vor dem Update mit dem Hinzufügen des Feldes **Briefanrede** zur Tabelle **tblAnreden** wiederhergestellt. Die Datensätze der Tabelle bleiben allerdings erhalten.

Automatisches Update bei Änderungen der Entitäten

```
PM> Enable-Migrations -EnableAutomaticMigrations
```

```
-> ASP:
https://weblogs.asp.net/senthil/code-first-ef-core
```

Update bei der Weitergabe

Wie aber sieht es nun aus, wenn wir das Projekt erstellen und auf die Zielrechner verteilen – werden die im Verzeichnis **Migrations** gespeicherten Klassen und ihre Methoden dann automatisch ausgeführt, wenn die Anwendung das erste Mal versucht, auf die Daten der Datenbank zuzugreifen? Genau genommen sollte die Anwendung, wenn sie das erste Mal auf

Adressen verwalten

Für unsere Lösung zum Erstellen und Verwalten von Word-Dokumenten aus dem Artikel »Briefe mit Word erstellen« benötigen wir eine Möglichkeit zum Anlegen und Verwalten von Adressen. Dazu wollen wir ein eigenes Fenster erstellen, dass eine Übersicht der Adressen in einer Liste, eine kleine Suchfunktion sowie die Details der aktuell in der Liste ausgewählten Adresse oder einer neuen Adresse anzeigt. Die Lösung soll natürlich über ein Entity Data Model an eine entsprechende Tabelle gebunden werden.

Zur Anzeige der Adressen fügen wir der Lösung ein neues Fenster namens **Adressen.xaml** hinzu. Für die Anordnung der benötigten Steuerelemente wollen wir dem Grid ein Raster von vier Spalten und acht Zeilen hinzufügen. Die erste und die dritte Spalte weisen als Spaltenbreite den Wert **Auto** auf, damit sich die Breite an den breitesten enthaltenen Steuerelementen orientiert, in diesem Fall an den Inhalten der **Label**-Elemente mit den Feldbeschriftungen. Die zweite und die vierte Spalte sollen sich den Rest der Breite aufteilen, damit der Benutzer das Fenster verbreitern und somit Inhalte der Textfelder, die gegebenenfalls nicht vollständig angezeigt werden, sichtbar machen kann.

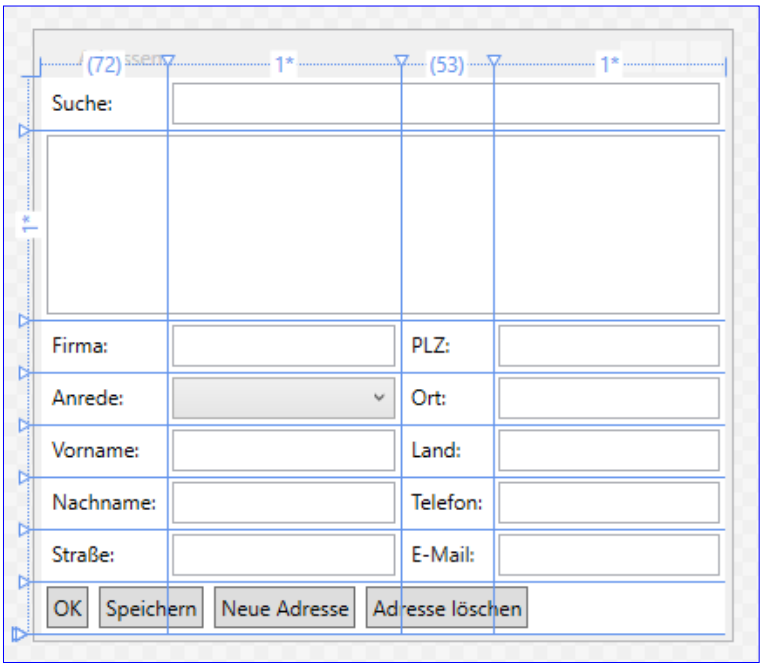


Bild 1: Fenster zum Verwalten von Adressen

Die Zeilen werden mit einer Ausnahme alle mit der Zeilenhöhe **Auto** ausgestattet, damit sich diese an der für die enthaltenen Elemente notwendigen Höhe orientieren können. Die Ausnahme ist die zweite Zeile, welche das Listenfeld zur Anzeige der Adressen in der Übersicht anzeigen soll. Diese Zeile erhält für die Eigenschaft **Height** den Wert *, was dazu führt, dass sie den verbleibenden Platz einnimmt und beim Vergrößern der Höhe des Fensters ebenfalls vergrößert wird. Die Definition des Grundgerüsts des Fensters mit XAML-Code sieht wie folgt aus und erscheint im Entwurf wie in Bild 1:

```
<Window x:Class="Adressen" ... Title="Adressen" Height="350" Width="400">
    <Window.Resources>
        <Style TargetType="TextBox">
            <Setter Property="Margin" Value="3"></Setter>
            <Setter Property="Padding" Value="3"></Setter>
        </Style>
        ...
    </Window.Resources>
```

```
    <Grid>
        <Grid.RowDefinitions>
            <RowDefinition Height="Auto"></RowDefinition>
            <RowDefinition Height="*"></RowDefinition>
            <RowDefinition Height="Auto"></RowDefinition>
            ...
        </Grid.RowDefinitions>
        <Grid.ColumnDefinitions>
            <ColumnDefinition Width="Auto"></ColumnDefinition>
            <ColumnDefinition Width="*"></ColumnDefinition>
            <ColumnDefinition Width="Auto"></ColumnDefinition>
            <ColumnDefinition Width="*"></ColumnDefinition>
        </Grid.ColumnDefinitions>
        <Label>Suche:</Label>
        <Label Grid.Row="2">Firma:</Label>
        <Label Grid.Row="3">Anrede:</Label>
        ... weitere Label
        <Label Grid.Row="2" Grid.Column="2">PLZ:</Label>
        <Label Grid.Row="3" Grid.Column="2">Ort:</Label>
        ... weitere Label
        <TextBox x:Name="txtSuchbegriff" Grid.Column="1" Grid.ColumnSpan="3"></TextBox>
        <ListBox x:Name="lstAdressen" Grid.Row="1" Grid.ColumnSpan="4" Margin="3" Padding="3"></ListBox>
        <TextBox x:Name="txtFirma" Grid.Row="2" Grid.Column="1"></TextBox>
        <ComboBox x:Name="cboAnreden" Grid.Row="3" Grid.Column="1"></ComboBox>
        <TextBox x:Name="txtVorname" Grid.Row="4" Grid.Column="1"></TextBox>
        ... weitere TextBox-Elemente
        <StackPanel Orientation="Horizontal" Grid.Row="7" Grid.ColumnSpan="4">
            <Button x:Name="btnOK">OK</Button>
            <Button x:Name="btnSpeichern">Speichern</Button>
            <Button x:Name="btnNeu">Neue Adresse</Button>
            <Button x:Name="btnLoeschen">Adresse löschen</Button>
        </StackPanel>
    </Grid>
</Window>
```

Im unteren Bereich findet sich noch ein **StackPanel**, welches ebenfalls über die gesamten vier Spalten geht und in horizontaler Ausrichtung vier **Button**-Elemente anzeigt. Für die Steuerelementtypen **Label, TextBox**, **ListBox**, **ComboBox** und **Button** haben wir im oberen Bereich unter **Window.Resources** einige allgemeine Eigenschaften wie **Margin** und **Padding** festgelegt. Diese greifen bei den Steuerelementen, die per **ColumnSpan** über mehrere Spalten angelegt werden, nicht – daher müssen wir zum Beispiel die Eigenschaft **Margin** für das Textfeld zur Eingabe des Suchbegriffs und das Listenfeld direkt mit dem Steuerelement definieren.

Entity Data Model

Das Entity Data Model für die Lösung haben wir bereits im Beitrag **EDM: Die Code First-Methode** und **EDM: Code First - Datenbank erweitern** besprochen. Der Teil, der für die hier zu verwaltenden Adressen verantwortlich ist, sind die beiden Klassen **Adresse** und **Anrede**. Wir benötigen für das Listenfeld eine Auflistung der Adressen mit den wichtigsten Feldern, für das Kombinationsfeld **cboAnreden** eine Auflistung der Anreden und für die Details der aktuell bearbeiteten Adresse ein Objekt auf Basis der Klasse **Adresse**. Um die Steuerelemente des Fensters an die Daten der beiden Klassen zu binden, fügen wir zunächst eine öffentliche Eigenschaft für ein Objekt auf Basis der Klasse Adresse zur Code behind-Klasse hinzu:

```
Public Property AktuelleAdresse As Adresse
```

Außerdem benötigen wir eine Variable zum Speichern der Objektvariablen für das **DbContext**-Element:

```
Private WordbriefContext As WordbriefEntities
```

Für die Liste der Adressen im **ListView**-Steuerelement sehen wir die folgende Variable vor:

```
Public Property Adressenliste As ObservableCollection(Of Adresse)
```

Fehlt noch eine Collection für die Anreden, die wir so deklarieren:

```
Public Property Anredenliste As ObservableCollection(Of Anrede)
```

Dann erstellen wir das Grundgerüst für die Konstruktor-Methode, also die Methode, die beim Öffnen des Fensters ausgelöst wird. Nun können wir das Fenster auf zwei Arten öffnen: Erstens mit der Schaltfläche zum Bearbeiten des aktuell im Hauptfenster **MainWindow.xaml** angezeigten Datensatzes und zweitens mit der Schaltfläche zum Erstellen eines neuen Adress-Objekts. Das heißt also, dass wir entweder eine ID aus dem aufrufenden Fenster übergeben müssen oder, wenn ein neues Element angelegt werden soll, eben nicht. Das ist eine gute Gelegenheit, einmal das Überladen der Konstruktor-Methode zu demonstrieren. Das bedeutet, dass wir zwei verschiedene Konstruktor-Methoden erstellen, von denen die eine keinen Parameter enthält – das ist die zum Anlegen eines neuen Elements. Die andere soll einen bestehenden Adress-Datensatz anzeigen, also übergeben wir dieser Konstruktor-Methode die ID des anzuzeigenden Datensatzes.

Dazu legen wir einen entsprechenden Parameter an. Grundsätzlich sehen die beiden Methoden also wie folgt aus:

```
Public Sub New()
    InitializeComponent()
    ...
End Sub
Public Sub New(AdresseID As Integer)
    InitializeComponent()
    ...
End Sub
```

Sie brauchen übrigens immer nur den Kopf mit **Public Sub New(...)** einzugeben, Visual Studio erzeugt dann immer den Rest mit der Anweisung **InitializeComponent** und den üblichen Kommentarzeilen – das gilt auch, wenn Sie Überladungen der **New**-Methode angeben.

Konstruktor-Methoden füllen

Damit begeben wir uns an die Konstruktor-Methoden. Die Methode **New** ohne Parameter soll dafür sorgen, dass das Fenster **AdressenVerwalten.xaml** einen neuen, leeren Datensatz anzeigt, den der Benutzer dann mit Daten füllen und speichern kann. Dazu benötigen wir die folgende Konstruktor-Methode:

```
Public Sub New()
    InitializeComponent()
    WordbriefContext = New WordbriefEntities
    Adressenliste = New ObservableCollection(Of Adresse)(WordbriefContext.Adressen)
    AktuelleAdresse = New Adresse
    Anredenliste = New ObservableCollection(Of Anrede)(WordbriefContext.Anreden)
    DataContext = Me
End Sub
```

Die Methode füllt **WordbriefContext** mit dem Entity Data Model. Außerdem liest sie die Adressen und Anreden in die Collections **Adressen** und **Anreden** ein. Die Variable **Adresse** wird einem neuen Element des Typs **Adresse** gefüllt. Die zweite Konstruktor-Methode erwartet den Parameter **SelektierteAdresse** des Typs **Adresse**, der vom aufrufenden Fenster übergeben wird und in den Steuerelementen des Fensters **AdressenVerwalten** als aktive Adresse angezeigt werden soll:

```
Public Sub New(selektierteAdresse As Adresse)
    InitializeComponent()
    WordbriefContext = New WordbriefEntities
    Adressenliste = New ObservableCollection(Of Adresse)(WordbriefContext.Adressen)
    Anredenliste = New ObservableCollection(Of Anrede)(WordbriefContext.Anreden)
    If Not selektierteAdresse Is Nothing Then
        AktuelleAdresse = selektierteAdresse
        lvwAdressen.SelectedValue = AktuelleAdresse.ID
    End If
    DataContext = Me
End Sub
```

Diese Methode unterscheidet sich nicht nur durch den Parameter von der ersten Konstruktor-Methode, sondern auch durch eine zusätzliche **If...Then**-Bedingung.

Diese prüft, ob der Parameter **selektierteAdresse** einen Wert enthält. Falls ja, wird die Eigenschaft **AktuelleAdresse** auf diese Adresse eingestellt. Außerdem soll das **ListView**-Steuerelement, dass ja auch mit der Liste der **Adress**-Objekte gefüllt wird, das im Parameter übergebene Element selektieren.

Bindungen zum XAML-Code hinzufügen

Nun haben wir die benötigten öffentlichen Eigenschaften in der Code behind-Klasse bereitgestellt, welche die Daten für die Steuerelemente des Fensters liefern sollen. Dementsprechend müssen wir nun die passenden Bindungen zu den Elementen hinzufügen. Als Erstes schauen wir uns dabei das **ListView**-Steuerelement an, dass wir auszugsweise wie folgt definieren:

```
<ListView x:Name="lvwAdressen" ItemsSource="{Binding Adressenliste}" SelectedValuePath="ID" Grid.Row="1" Grid.Column-
Span="4" Margin="3" Padding="3" SelectionChanged="lvwAdressen_SelectionChanged">
    <ListView.Resources>
       <Style TargetType="GridViewColumnHeader">
            <Setter Property="HorizontalContentAlignment" Value="Left"></Setter>
        </Style>
    </ListView.Resources>
    <ListView.View>
        <GridView>
            <GridView.Columns>
                <GridViewColumn DisplayMemberBinding="{Binding Firma}">
                    <GridViewColumn.Header>
                        <GridViewColumnHeader Tag="Firma">Firma</GridViewColumnHeader>
                    </GridViewColumn.Header>
                </GridViewColumn>
                <GridViewColumn DisplayMemberBinding="{Binding Vorname}">
                    <GridViewColumn.Header>
                        <GridViewColumnHeader Tag="Vorname">Vorname</GridViewColumnHeader>
                    </GridViewColumn.Header>
                </GridViewColumn>
            </GridView.Columns>
        </GridView>
    </ListView.View>
```

AUSGABE 02-03/2018

DATENBANK ENTWICKLER

MAGAZIN FÜR DIE DATENBANKENTWICKLUNG MIT VISUAL STUDIO FÜR DESKTOP, WEB UND CO.

TOP-THEMEN:

VISUAL STUDIO	Visual Studio 2017 Community	SEITE 3
ASP.NET CORE	Webanwendungen erstellen	SEITE 8
ASP.NET CORE	Webdesign mit Bootstrap Studio	SEITE 29
INTERAKTIV	E-Mails per Sendgrid verschicken	SEITE 41
ASP.NET CORE	Razor Pages mit Datenbankanbindung	SEITE 70

André Minhorst Verlag

Visual Studio 2017 Community Edition

Nachdem die Beispiele der bisherigen Ausgaben von DATENBANKENTWICKLER, also bis Ausgabe 1/2018, mit Visual Studio 2015 erstellt wurden, steigen wir mit Ausgabe 2/2018 auf Visual Studio 2017 um. Dieser Artikel beschreibt die gegenüber der Version Visual Studio 2015 leicht veränderte Installation. Außerdem fügen wir gleich noch ein paar Komponenten hinzu, mit denen wir uns auf neues Terrain begeben – der Entwicklung einfacher Webanwendungen.

Zum Download der Visual Studio 2017 Community Edition gelangen Sie über den Link **https://www.visualstudio.com/de/**. Dort wechseln Sie zum Download **Visual Studio-IDE** und wählen dort unter **Für Windows herunterladen den Eintrag Community 2017** aus (siehe Bild 1).

Bild 1: Download von **Visual Studio 2017 Community**

Dies lädt eine kleine Datei herunter, die Sie starten und die weitere Informationen nachlädt. Danach erscheint der Dialog aus Bild 2. Wer weiß, warum wir schon beim Download die Edition auswählen mussten? Vermutlich noch nicht einmal Microsoft selbst.

Jedenfalls erhalten wir nach dem Start des Setup-Programms nochmal die Auswahl der gewünschten Edition. Wiederum wählen wir den Eintrag **Visual Studio Community 2017** aus.

Damit landen wir in einem weiteren Dialog, in dem wir einfach auf **Installieren** klicken könnten (siehe Bild 3).

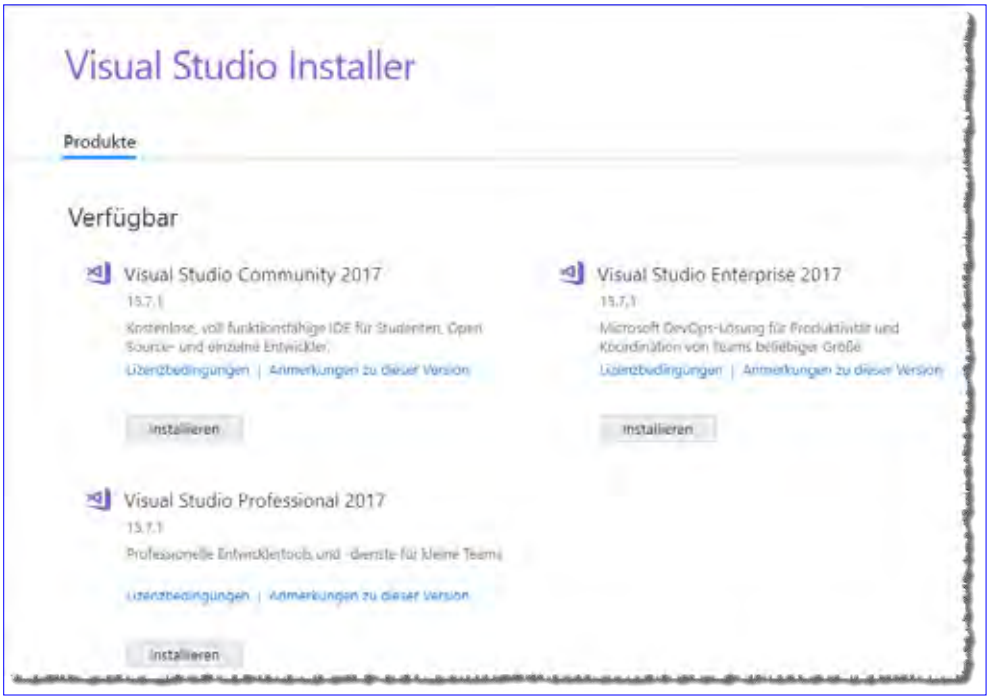


Bild 2: Start des Installers

Wenn wir dies tun, erscheint allerdings die Meldung aus Bild 4. Ohne ein sogenanntes **Workload**-Element auszuwählen, würde das Setup nur den Kern-Editor installieren, der aber keine Tools für den Umgang mit Projekten liefert.

Also schauen wir uns doch einmal genauer an, welche Optionen uns der Bereich **Workloads** bietet.

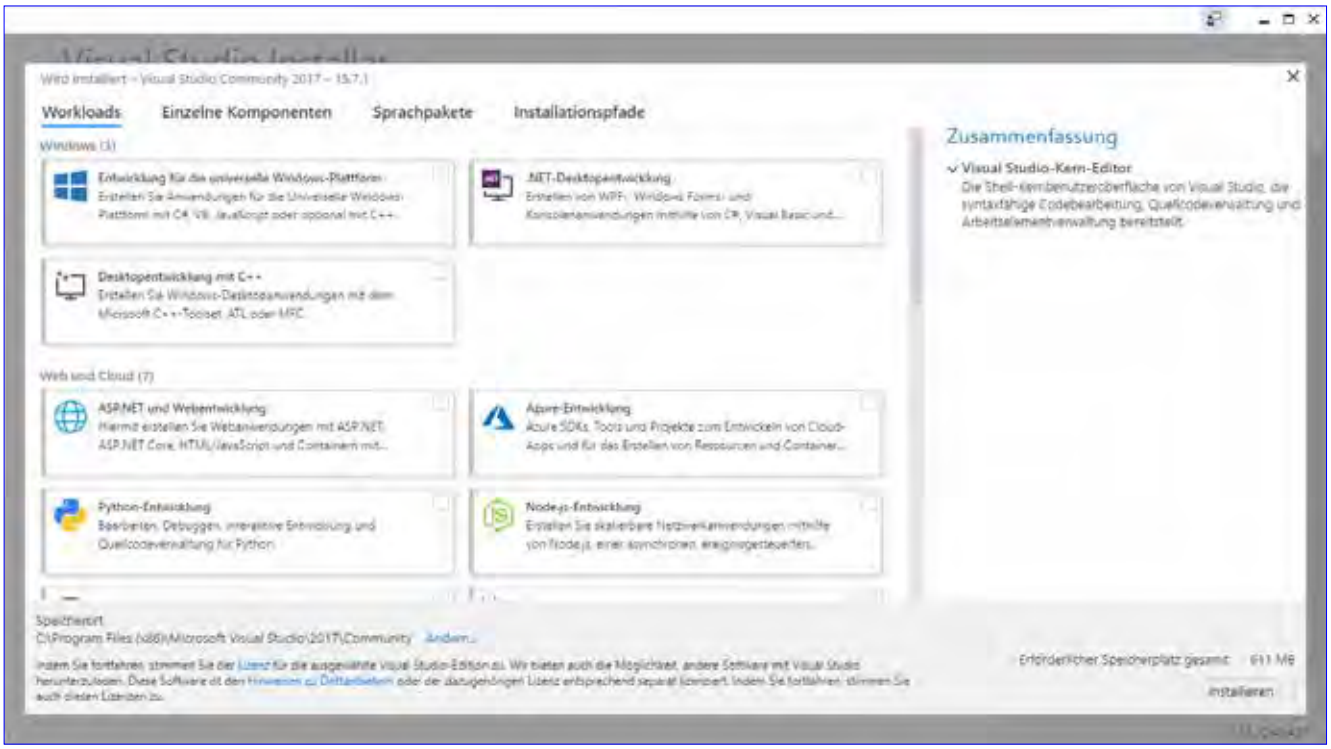


Bild 3: Auswahl der Workloads

Als Erstes aktivieren wir den Eintrag **.NET Desktopentwicklung**, denn diesen benötigen wir zur Entwicklung von WPF-Anwendungen mit den verschiedenen Sprachen. Sobald wir diese Option anhaken, erhalten wir im rechten Bereich noch eine Zusammenfassung der installierten Optionen. Einige davon sind standardmäßig installiert, weitere können Sie optional hinzufügen. Wir nehmen hier zumindest noch den Eintrag **SQL Server Express 2016 LocalDB** hinzu.

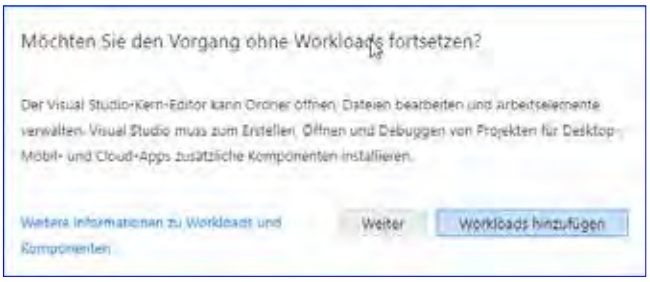


Bild 4: Meldung, wenn man keine Workloads installieren möchte

Außerdem wollen wir auch gleich noch die **.NET Core 1.0 - 1.1-Entwicklungstools** und die **.NET Core 2.0-Entwicklungstools** mitnehmen. .NET Core soll eine neue, leichtgewichtigere Implementierung des .NET-Frameworks sein, das plattformunabhängig und modular aufgebaut ist. Allerdings ist es noch nicht vollständig implementiert und liefert so beispielsweise noch keine Unterstützung für die Entwicklung von WPF-Anwendungen. Sie können damit aber Konsolen- und Webanwendungen mit den neuesten Technologien entwickeln. Das ist auch der Grund, warum wir die .NET Core-Elemente beim Setup mit hinzunehmen.

Das nächste **Workload**-Element, welches wir hinzufügen wollen, ist das Element **ASP.NET und Webentwicklung**. Auch hier gibt es wieder einige vordefinierte Unterelemente im rechten Bereich. Hier nehmen wir noch die beiden Einträge **Entwicklungszeit-IIS-Unterstützung** mit sowie **.NET Core 1.0 - 1.1-Entwicklungstools für Web** mit. Im Zusammenhang damit fügen wir auch noch das **Workload**-Element **Node.js-Entwicklung** hinzu.

Interpolierte Zeichenketten

Beim Programmieren von Datenbankanwendungen arbeiten Sie immer wieder mit Zeichenfolgen. Dabei setzen Sie Zeichenketten mitunter auch aus mehreren Teilzeichenketten und den Werten aus Variablen oder Funktionen zusammen. Um dies zu erledigen, gibt es verschiedene Vorgehensweisen, die mehr oder weniger gut lesbar sind. Dieser Artikel zeigt die Möglichkeiten zum Zusammensetzen von Zeichenketten.

Herkömmliches Zusammensetzen von Zeichenketten

Wer von VBA kommt, kennt es nicht anders: Zeichenketten, die miteinander verknüpft werden, verbindet man mit dem entsprechenden Operator, hier zum Beispiel dem Kaufmanns-Und (&):

```
Dim strUnd As String
strUnd = " und "
Debug.Print "A" & strUnd & "B"
```

Heraus kommt dann die Zeichenfolge **A und B**. Unter C# können Sie das grundsätzlich genauso machen, wobei Sie hier allerdings den Plus-Operator verwenden und nicht das Kaufmanns-Und:

```
string und = "und";
string oder = "oder";
Console.WriteLine("A " + und + " B");
```

Das Ergebnis ist das gleiche. Die Übersichtlichkeit leidet hier noch nicht allzu sehr, wenn Sie jedoch mehrere Zeichenketten zusammenführen wollen, geschieht dies recht schnell:

```
Console.WriteLine("((A " + und + " B) " + oder + " (C " + und + " D))");
```

Die **WriteLine**-Methode ermöglicht es allerdings, das Zusammensetzen zu vereinfachen – und zwar durch den Einsatz eines speziellen Formats, bei dem Sie im Text in geschweifte Klammern eingefasste Zahlen einfügen, welche als Platzhalter für die im Anschluss aufgeführten Variablen dienen. Dies sieht dann wie folgt aus:

```
Console.WriteLine("((A {0} B) {1} (C {2} D))", und, oder, und);
```

Wenn Sie wie in diesem Fall zwei Mal die gleiche Variable verwenden, können Sie dies sogar noch vereinfachen und zwei Mal auf den Platzhalter **{0}** zugreifen:

```
Console.WriteLine("((A {0} B) {1} (C {0} D))", und, oder);
```

Alle drei vorherigen Ausdrücke liefern das folgende Ergebnis:

Einfache ASP.NET Core-Anwendung erstellen

ASP.NET-Webanwendungen gibt es schon, seit Microsoft .NET vorgestellt hat. Mit .NET Core und Bootstrap wird dies noch interessanter, vor allem für die Programmierung moderner Webanwendungen mit Responsive Webdesign – also mit einem Design, das auf die Gegebenheiten des jeweiligen Endgerätes reagiert und die Darstellung darauf optimiert. Dies wird möglich mit dem neuen ASP.NET Core und der Bootstrap-Bibliothek. Dieser Artikel zeigt, wie Sie eine einfache Anwendung auf Basis von ASP.NET Core mit Bootstrap erstellen.

Voraussetzungen

Voraussetzung für das Reproduzieren der in diesem Artikel vorgestellten Abläufe ist das Vorhandensein von Visual Studio Community 2017. Daneben stellt sich an dieser Stelle heraus, dass es gut war, in diesem Magazin zunächst die Programmiersprache C# vorzustellen. ASP.NET Core unterstützt nämlich aktuell noch kein Visual Basic, sondern nur C#.

Projekt anlegen

Um das neue Projekt anzulegen, starten Sie Visual Studio und öffnen den Dialog zum Erstellen eines neuen Projekts. Hier wählen Sie als Projektvorlage den Eintrag **Visual C#\|Web\|ASP.NET Core-Webanwendung** aus (siehe Bild 1).

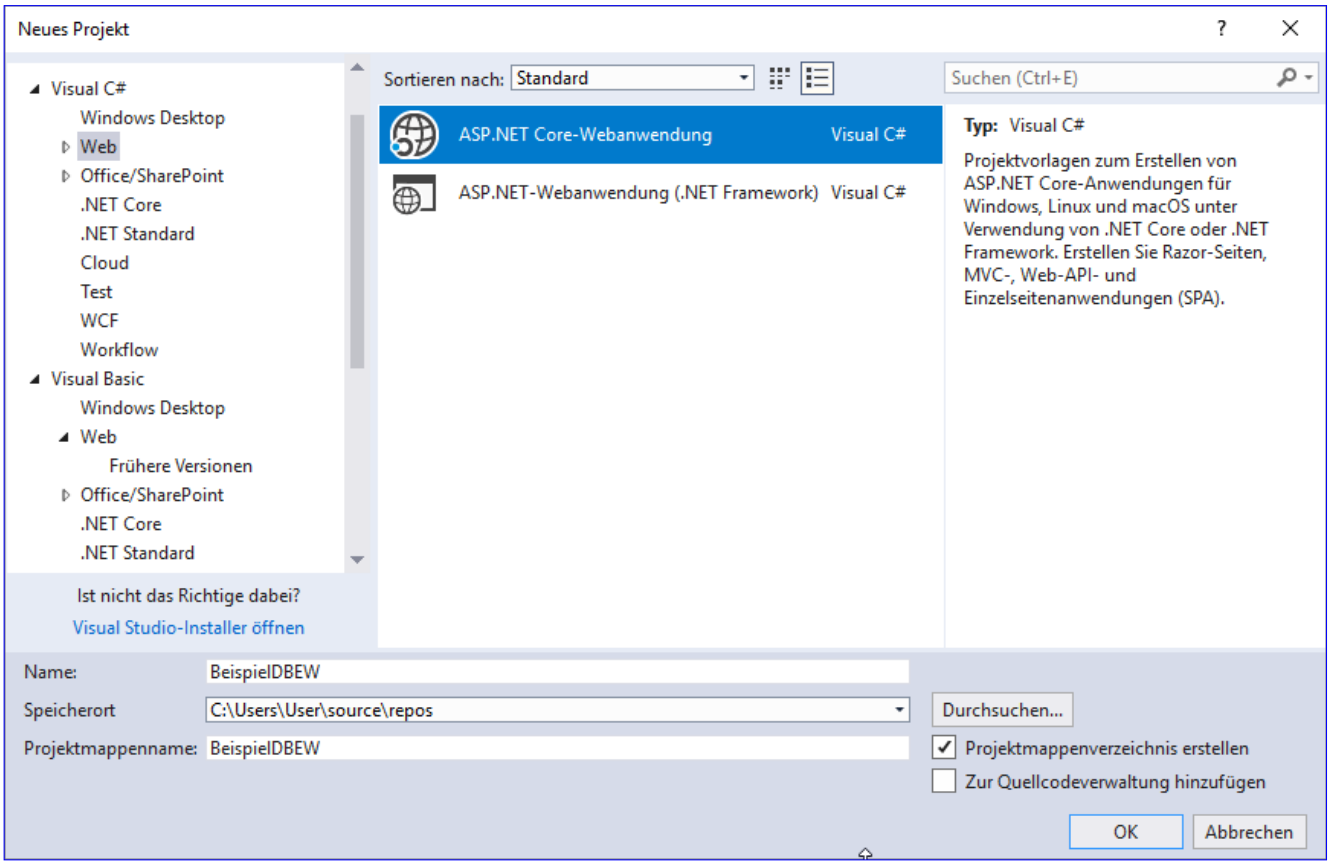


Bild 1: Erstellen einer Webanwendung für .NET Core

Nach dem Festlegen des gewünschten Projektnamens sowie des Speicherorts legt Visual Studio das Projekt allerdings noch nicht direkt an. Zuvor erscheint noch der Dialog aus Bild 2. Hier behalten wir den Eintrag **Webanwendung** sowie die übrigen Einstellungen bei und klicken auf die Schaltfläche **OK**.

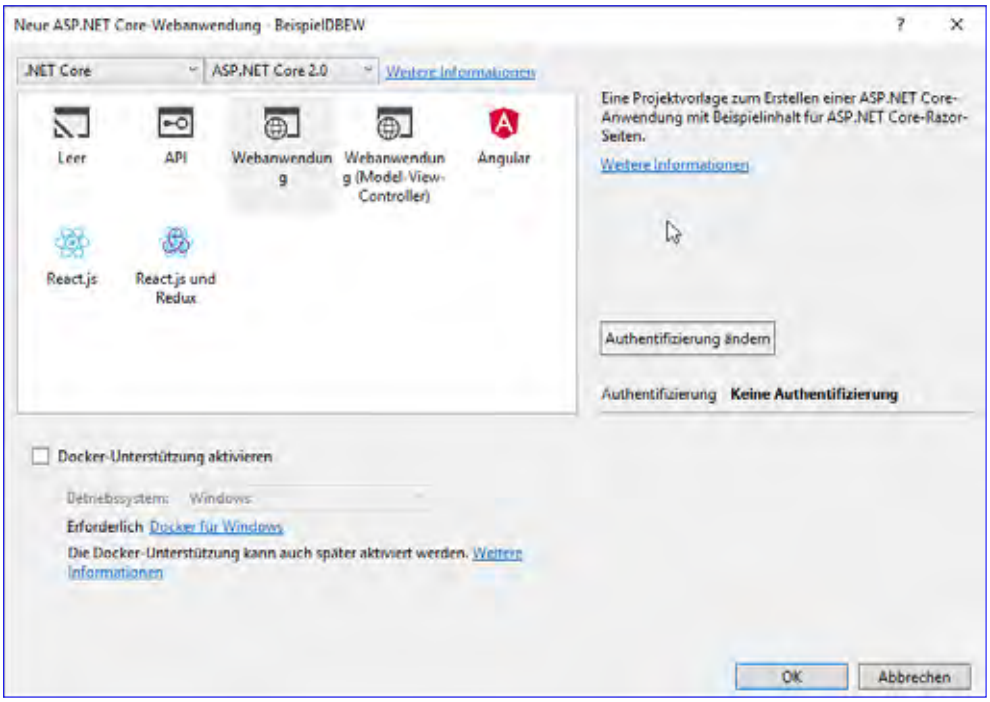


Bild 2: Festlegen weiterer Parameter für die neue Weblösung

Anschließend wird das Projekt erstellt. Ein Blick in den Projektmappen-Explorer liefert die erstellten Projektdateien, die sich – wenig überraschend – in ihrer Zusammensetzung weitgehend von denen einer Desktop-Anwendung unterscheiden (siehe Bild 3). Interessant ist für uns zunächst der Ordner **wwwroot**, der **.css**-Dateien, Bilder und weitere Dateien enthält. Die eigentlichen Internetseiten werden im Verzeichnis **Pages** gespeichert. Hier finden Sie die bekannte Bezeichnungen wie **Index**, **Contakt**, **About** und so weiter, allerdings nicht mit einer Dateiendung wie **.html**, sondern mit der Dateiendung **.cshtml**.

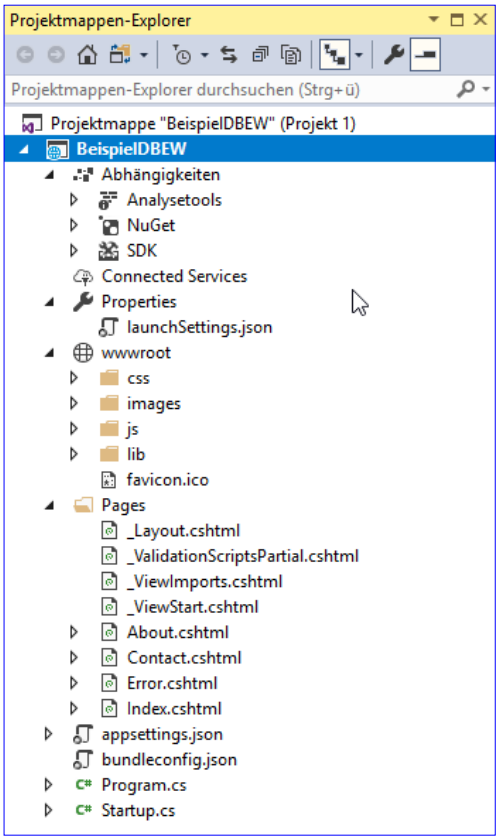


Bild 3: Projektmappen-Explorer der Webanwendung

ASP.NET Core-Projekt starten

Bevor wir näher auf die einzelnen Dateien eingehen, sind wir neugierig, was geschieht, wenn wir einfach einmal auf die **Starten**-Schaltfläche klicken. Diese enthält in diesem Fall die Beschriftung **IIS Express**, was darauf hindeutet, dass wir uns mit der neuen Version von Visual Studio auch gleich noch einen Internet Informationen Server in der Express-Variante installiert haben. Ein Klick auf die **Starten**-Schaltfläche blendet dann in Visual Studio die Diagnose-Tools ein, welche die Speicherauslastung anzeigen. Interessanter ist aber, dass die Anwendung in einem neuen Browser-Fenster angezeigt wird (sofern nicht bereits ein Browser geöffnet ist). Hier finden wir bereits einen Header-Bereich mit der Seitennavigation sowie ein Slide-Element, das verschiedene Features für Web-Anwendungen anpreist (siehe Bild 4). Die Links funktionieren wir erwartet und liefern die entsprechenden weiterführenden Seiten im Browser.

In der URL-Leiste finden wir eine Adresse wie **localhost:12345**. **localhost** ist der Standardhostname für den lokalen Computer. Hinter dem Doppelpunkt folgt die Angabe des Ports in Form einer fünfstelligen Zahl. Diese Zahl wird beim Starten des Projekts zufällig vergeben. Unter **Application uses** finden wir bereits einen Hinweis auf die verwendeten Techniken. **ASP.NET Core Razor Pages** bieten einen einfacheren Ansatz zur Programmierung von Web-Anwendungen als es mit den bisherigen ASP.NET basierten Anwendungen nach dem MVC-Pattern der Fall war. Die zweite Technik heißt **Bootstrap**. Dabei handelt es sich um eine Bibliothek, welche das Design von Anwendungen für verschiedene Endgeräte vereinfacht. Wenn Sie unter Verwendung von Bootstrap nun beispielsweise das Browser-Fenster viel schmaler machen, verschwinden die Einträge der Navigationsleiste und werden in Form eines kleinen Icons rechts oben angezeigt, welches dann per Mausklick die Menüeinträge liefert (siehe Bild 5).

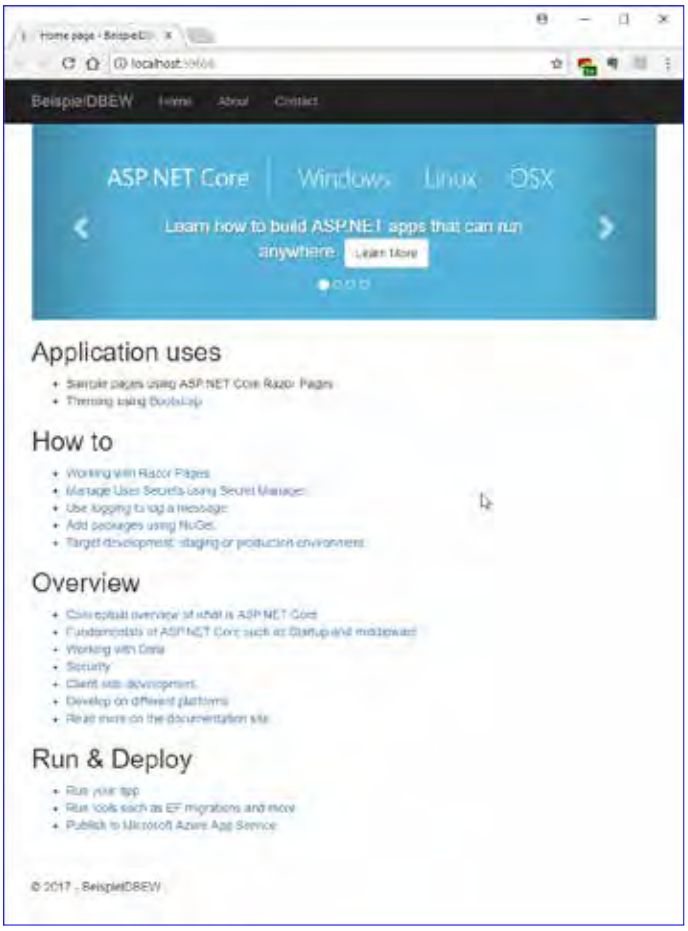


Bild 4: Erster Start der Beispielanwendung

Damit sind sowohl Bootstrap also auch **Razor Pages** bereits in der Beispielanwendung enthalten. **Razor Pages** war die Vorlage, die wir gewählt haben, also sind die entsprechenden Elemente ebenfalls bereits vorhanden. Mit der Beispielanwendung können Sie nun ein wenig experimentieren. Anschließend legen wir ein neues, leeres ASP.NET Core-Projekt an und zeigen, wie Sie selbst die benötigten Elemente hinzufügen.

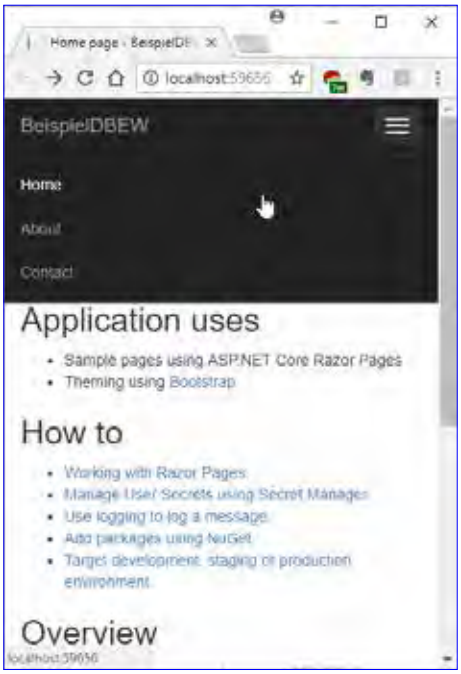


Bild 5: Vorteil mit Bootstrap: Skalierung der Größe der Anwendung

Leeres Projekt anlegen

Ein leeres Projekt legen Sie auf ähnliche Weise an. Im Dialog **Neues Projekt** wählen Sie wiederum den Eintrag **ASP.NET Core-Webanwendung** aus und geben den Projektnamen und den Pfad an – in diesem Fall verwenden wir **DBEWBlank**. Im danach erscheinenden Dialog **Neue ASP.NET Core-Webanwendung** wählen Sie die Vorlage **Leer** aus (siehe Bild 6).

Wenn Sie das Projekt schon jetzt starten, erhalten Sie im Browser die Ansicht aus Bild 7. Woher kommt diese Seite? Dazu schauen wir uns den Code der Datei **Startup.cs** an, der nach dem Entfernen der Kommentare wie folgt aussieht.

ASP.NET Core-Anwendung anpassen

Wenn Sie ein neues Projekt auf Basis der Vorlage »Visual C#|Web|ASP.NET Core-Webanwendung« mit dem Typ »Webanwendung« erstellt haben, finden Sie eine komplette Beispielanwendung vor. Im Gegensatz dazu steht der Typ »Leer«, mit dem Sie eine komplett leere Anwendung erstellen, der allerdings auch jegliche Infrastruktur wie etwa das Menü, die CSS-Dateien und vieles mehr fehlen. Wir wollen einmal von einer Anwendung des Typs »Webanwendung« ausgehend betrachten, welche Anpassungen notwendig sind, um die Beispielinhalte, -menüs und -designs in eigene Elemente umzuwandeln.

Für die Beispiele dieses Artikels legen Sie wieder ein Projekt auf Basis der Vorlage **Visual C#|Web|ASP.NET Core-Webanwendung** an. Wählen Sie im Dialog **Neue ASP.NET Core Webanwendung** den Eintrag **Webanwendung** aus. Dies liefert eine Anwendung, die nach dem Starten ein Menü und entsprechende Inhalte im Webbrowser anzeigt (siehe Bild 1). Allerdings ist es gar nicht so einfach, zu identifizieren, wo die einzelnen Bestandteile der Anwendung wie die Inhalte, das Menü und das Layout definiert sind. Somit ist auch die Anpassung etwas kompliziert. Aber genau das beleuchtet der vorliegende Artikel: Wir sehen uns an, wo die verschiedenen Elemente, die Sie nach dem Start der Beispielanwendung sehen, angepasst werden können und wie Sie eine für eigene Zwecke vorbereitete Vorlage daraus erstellen können.

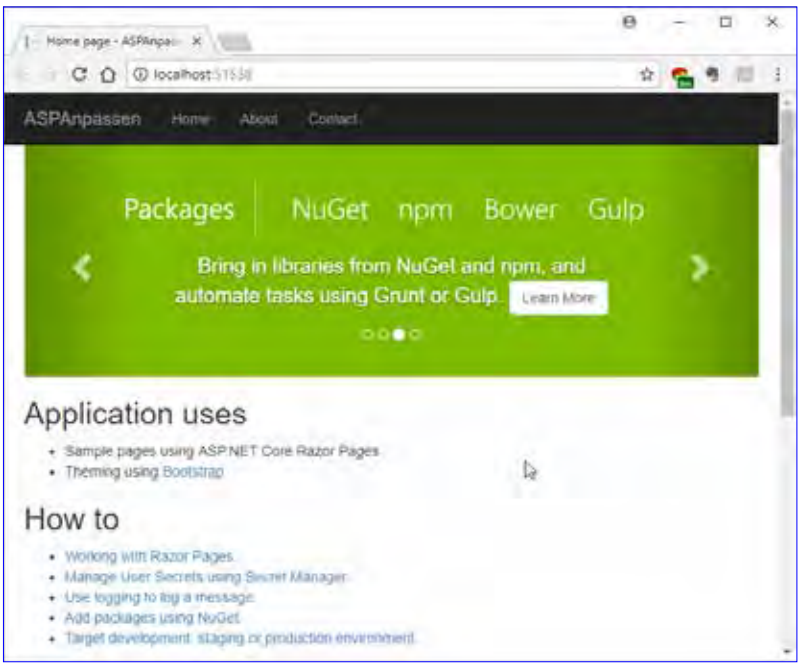


Bild 1: So sieht die Anwendung nach dem ersten Start aus.

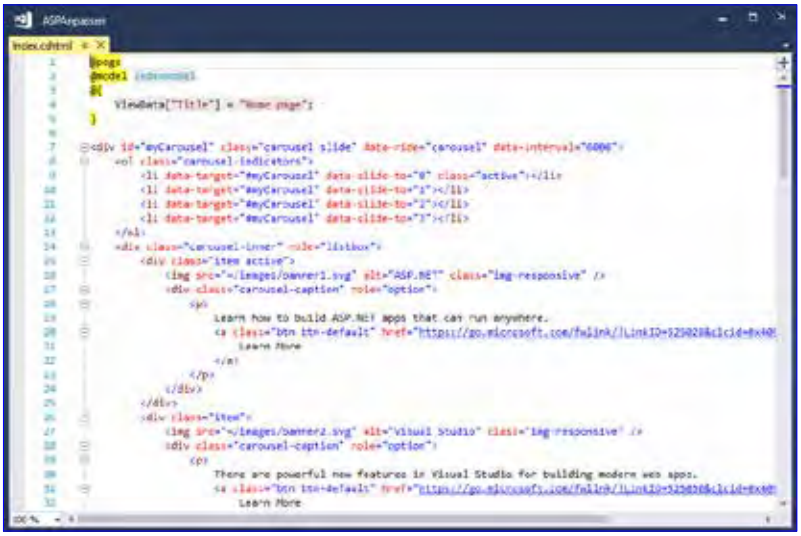


Bild 2: Code der Seite **Index.cshtml**

Startseite anpassen

Wir beginnen mit der Seite, die zu Beginn angezeigt wird. In einem Projekt, das auf Basis der oben genannten Vorlage erstellt wurde, sind die Seiten mit den Inhalten im Ordner **Pages** untergebracht. Hier finden wir auch gleich die Seite **Index.cshtml**. Die Index-Seite ist normalerweise die zuerst angezeigte Seite einer Webanwendung,

wenn keine anderen Seite in der URL angegeben wurde. Nach einem Doppelklick auf den entsprechenden Eintrag im Projektmappen-Explorer zeigt dann auch gleich den Code der Seite an (siehe Bild 2). Interessanterweise finden wir hier ganz oben nicht etwa den Code für die Menüleiste, sondern den für den Slider, der durchlaufend verschiedene Inhalte anzeigt. Um zu prüfen, welche Inhalte nun überhaupt in der Seite **Index.cshtml** gespeichert werden, kommentieren wir einfach einmal alle HTML-Inhalte dieser Seite aus. Das erledigen Sie in **.cshtml**-Seiten auf eine der folgenden Arten: Entweder Sie verwenden die unter HTML üblichen Elemente zum Auskommentieren (**<!--...-->**) oder Sie nutzen die dafür vorgesehenen Tags der Razor Pages, nämlich **@*...*@**. Ich nutze gern immer die zum Auskommentieren vorgesehene Tastenkombination **Strg + K, Strg + C** und zum Einkommentieren **Strg + K, Strg + U**. Dies schließt den markierten Text in **@*...*@** ein und kommentiert diesen so aus.

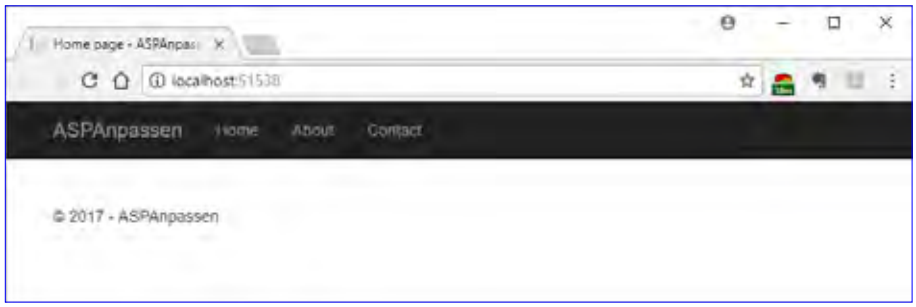


Bild 3: Was von **Index.cshtml** übrig bleibt ...

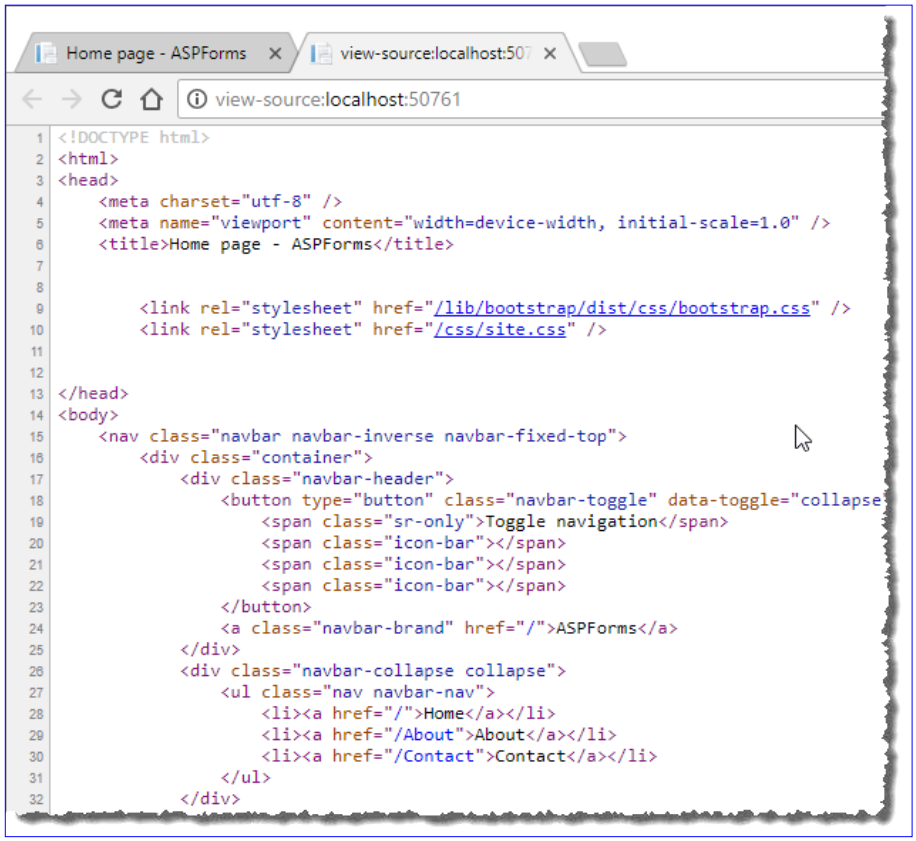


Bild 4: Quellcode der Startseite

Danach starten wir die Webanwendung erneut und schauen uns an, was von der Startseite übrig bleibt. Das Ergebnis sehen Sie in Bild 3 – es bleiben nur das Menü und die Fußzeile übrig. Damit wissen wir schon einmal, welchen Teil der Webanwendung wir über die Seite **Index.cshtml** manipulieren können. Darüber hinaus stellt sich heraus, dass **Index.cshtml** nicht den kompletten Inhalt der angezeigten Startseite enthält, sondern nur den zwischen Navigationsleiste und Fußzeile erscheinenden Inhalt. Den Rest müssen wir also an anderen Stellen im Projekt suchen.

Navigationsleiste ändern

Als Nächstes wollen wir herausfinden, wo sich der Code für die Navigationsleiste befindet. Dazu schauen wir uns den Quellcode der Webseite an. Dazu klicken Sie beispielsweise unter Google Chrome mit der rechten Maustaste auf die Webseite und wählen den Kontextmenü-Eintrag **Seitenquelltext anzeigen** aus. Dies öffnet dann eine neue Registerseite im Browser mit

Markup mit Razor Pages

Wenn Sie Webseiten auf Basis von ASP.NET Core bauen, wollen Sie in den statischen HTML-Code gegebenenfalls auch dynamische Elemente einbauen. Dazu verwenden Sie eine sogenannte Server Side Markup Language. Eine solche Sprache ist das relativ neue Razor Pages. Dieser Artikel zeigt, wie Sie den Inhalt von HTML-Seiten über Razor Pages durch dynamische Inhalte erweitern.

Wenn Sie schon einmal eine Seite mit PHP oder mit klassischen ASP programmiert haben, kennen Sie die Code-Schnipsel, die dort in den HTML-Code eingepflegt werden. Unter PHP etwa beginnen diese mit **<?php** und enden mit **?>**. Für ASP.NET Core gibt es mit Razor Pages eine neue Technologie, die wir uns im vorliegenden Artikel ansehen wollen. Voraussetzung dafür ist, dass Sie ein einfaches Webprojekt auf Basis von ASP.NET Core angelegt haben – siehe auch im Beitrag **Einfache Web-Anwendung erstellen**. Die dort angelegte Anwendung nutzt eine statische Seite namens **index.html** als Startseite. Diese kann wegen ihrer Dateiendung keine dynamischen Elemente aufnehmen.

Model View Controller hinzufügen

Model View Controller ist ein Entwurfsmuster, mit dem eine Software in verschiedene Ebenen aufgeteilt wird. Das Model entspricht dem Datenmodell, die View ist die Präsentation der Daten, also die Benutzeroberfläche und der Controller steuert die Anwendung. An dieser Stelle steigen wir noch nicht tiefer in dieses Entwurfsmuster ein (genau genommen verwenden wir MVC noch nicht einmal im eigentlichen Sinne), aber wir benötigen einen Service, der uns bei der Ausführung sogenannter MVC-Aktionen unterstützt. In diesem Fall wird die MVC-Aktion eine HTML-Seite mit einigen Besonderheiten sein, mehr dazu weiter unten. Als Erstes schauen wir uns an, welche Änderungen im Quellcode der Klasse **Startup.cs** nötig sind, um die Verarbeitung von MVC-Aktionen zu unterstützen. Wir fügen also zunächst den Aufruf der Methode **AddMVC** zur Methode **ConfigureServices** hinzu. Danach erweitern wir die Methode **Configure** neben dem Aufruf der Methode **UseStaticFiles** um die Methode **UseMVC**:

```
public class Startup {
    public void ConfigureServices(IServiceCollection services) {
        services.AddMvc();
    }
    public void Configure(IApplicationBuilder app, IHostingEnvironment env) {
        if (env.IsDevelopment()) {
            app.UseDeveloperExceptionPage();
        }
        app.UseStaticFiles();
        app.UseMvc();
    }
}
```

Razor-Seite hinzufügen

Nun benötigen wir einen Ersatz für die bisher statisch verwendete Seite **index.html**. Dazu rufen wir wieder den Kontextmenü-Eintrag **Neues Element hinzufügen...** auf, diesmal aber für den noch hinzuzufügenden Ordner **Pages**. Im Dialog **Neues**

Element hinzufügen wählen Sie unter der Kategorie **Web** den Eintrag **Razor Seite** aus und legen den Namen **index.cshtml** für die Seite fest (siehe Bild 1).

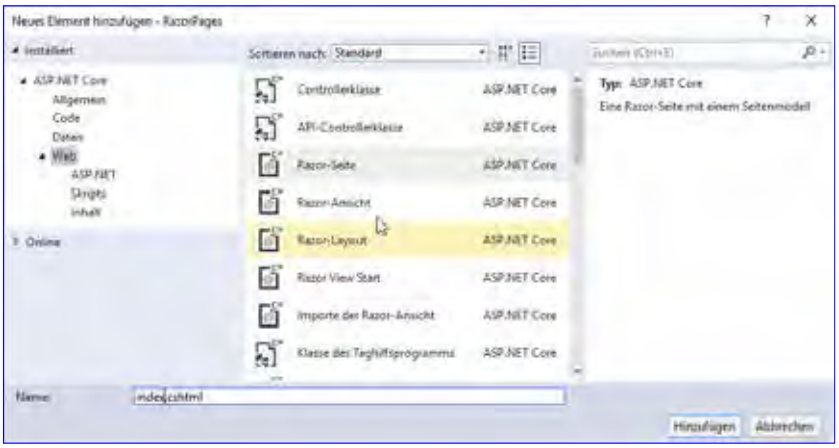


Bild 1: Hinzufügen einer Razor-Seite

Die neue Seite namens **index.cshtml** sieht wie in Bild 2 aus. Hier ist zunächst kein HTML-Code in Sicht, was ein wenig irritiert. Stattdessen erwartet uns ein als Fehler markierter Ausdruck namens **indexModel**.

Bild 2: Code der neuen Seite **index.cshtml**

Zusätzlich liefert der Projektmappen-Explorer noch eine der Seite **index.cshtml** untergeordnete Datei namens **index.cshtml.cs** (siehe Bild 3). Das Schema erinnert uns an die Code behind-Klassen von WPF-Seiten. Der Fehler hinter dem Element **indexModel** in der Datei **index.cshtml** lautet »Der Typ- oder Namespacename "indexModel" wurde nicht gefunden (möglicherweise fehlt eine using-Direktive oder ein Assemblyverweis).«.

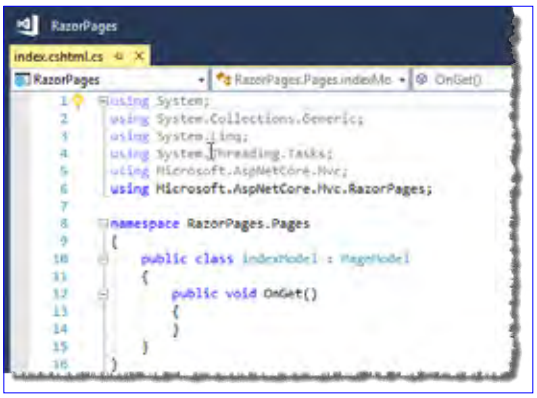


Bild 3: Die »Code behind«-Klasse von **index.cshtml**

Wenn wir die Anwendung starten, erhalten wir auch tatsächlich eine Fehlermeldung im Browser (siehe Bild 4). Das ist übrigens dem Umstand geschuldet, dass wir in der Methode **Configure** der Klasse **Startup** für den Entwicklungsmodus die Methode **UseDeveloperExceptionPage** aufgerufen haben. Der Fehler liegt darin, dass wir den Namespace zur Seite **index.cshtml** nicht angegeben haben. Das holen wir nun nach, indem wir den Code der Seite **index.cshtml** wie folgt ändern und den Namespace **RazorPages** sowie den Namen des Ordners **Pages** hinzufügen:

```
@page
@model RazorPages.Pages.indexModel
@{
}
```

Bild 4: Fehler beim Starten der Anwendung

Danach können wir die Anwendung ohne Fehlermeldung starten, erhalten allerdings auch eine leere Seite – kein Wunder, denn wir haben ja noch keine Inhalte auf der Seite **index.cshtml** hinterlegt. Das ändern wir nun durch Hinzufügen der folgenden Zeile:

```
<h1>Hello index.cshtml!</h1>
```

Beim nächsten Start der Anwendung sehen wir dann auch den gewünschten Text im Browser (siehe Bild 5).

Bild 5: Die Seite läuft!

Dynamische Inhalte hinzufügen

Nun gehen wir einen Schritt weiter und wollen einen dynamischen Inhalt hinzufügen – was der eigentliche Sinn des Artikels ist. Dazu nutzen wir das **@**-Zeichen und die dadurch per IntelliSense eingeblendeten verfügbaren Code-Elemente. In diesem Fall wollen wir das aktuelle Datum und die Uhrzeit ausgeben (siehe Bild 6). Das erledigen wir mit folgendem Code (die ersten Zeilen bleiben natürlich erhalten):

```
<h1>Hello index.cshtml!</h1>
<p>Es ist jetzt @DateTime.Now</p>
```

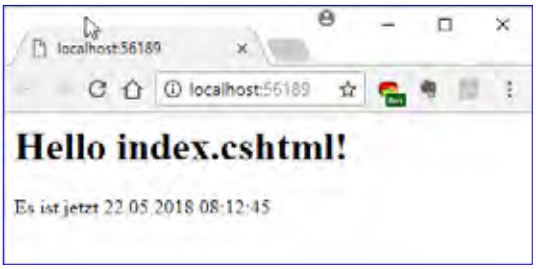


Bild 6: Anzeige von Datum und Uhrzeit

Razor-Anweisungen

Schauen wir uns nun an, was die mit **@** beginnenden Anweisungen in der Datei **index.cshtml** bedeuten.

- **@page**: Diese Anweisung kennzeichnet die Seite als sogenannte MVC-Aktion. Dies ist immer die erste Anweisung auf der Seite.

- **@model RazorPages.Pages.indexModel**: Diese Seite gibt an, wo sich das Model für diese Seite (View) befindet. Dabei handelt es sich um die Klasse **indexModel** im Ordner **Pages** des Namespaces **RazorPages**.

- **@{...}**: Hier können Sie C#-Anweisungen eingeben, die zur Laufzeit auf dem Server ausgeführt werden.

Möglichkeiten, C#-Code und HTML in Razor Pages zu kombinieren

Wenn Sie HTML-Code und C#-Code so kombinieren wollen, dass der C#-Code zur Laufzeit auf dem Server ausgeführt wird und das Ergebnis mit dem bereits vorhandenen HTML-Code zusammengestellt wird, haben Sie verschiedene Möglichkeiten:

- Sie schreiben den C#-Code in **@{...}**-Blöcke. Jede Anweisung muss dann mit einem Semikolon abgeschlossen werden.

- Sie fügen Variablen oder Funktionsaufrufe mit führendem **@** in den HTML-Code ein.

Die Dateien, die solche Kombinationen aus HTML- und C#-Code enthalten, müssen immer die Dateiendung **.cshtml** aufweisen. Auf diese Weise können Sie etwa in einem Razor Code-Block eine Variable deklarieren und füllen und dann im HTML-Code über den **@**-Operator auf die Variable zugreifen. Hier ist gleich anzumerken, dass Variablen in Razor-Blöcken immer als **var** deklariert werden. Anweisungen werden wie unter C# üblich mit dem Semikolon abgeschlossen (**;**). Zeichenketten werden in Anführungszeichen eingefasst. Achtung: Razor Code unter C# ist case sensitive, **Vorname** ist also ungleich **vorname**. Ein Beispiel für sieht etwa wie folgt aus:

```
@page
```

```
@model RazorPages.Pages.indexModel
@{
    var Seitentitel = "Startseite";
}
<h1>@Seitentitel</h1>
```

Dies liefert im Browser eine Seite, die lediglich den Text **Startseite** im Format **h1** enthält. Was geschieht nun auf dem Server? Dieser führt zunächst die Razor Code-Anweisungen aus, fügt den dadurch generierten Inhalt an den entsprechenden Stellen im HTML-Code ein und schickt dann beides zusammen zum Client.

Pfad zum Model per using

Die Anweisung **@model RazorPages.Pages.indexModel**, die das Model über den Namespace **RazorPages**, den Pfad **Pages** und den Klassennamen **indexModel** referenziert, können wir auch wie folgt auf zwei Anweisungen aufteilen:

```
@using RazorPages.Pages
@model indexModel
```

Das in der **@model**-Zeile angegebene Model können wir anschließend im HTML-Code mit **@Model** referenzieren – wobei Model groß geschrieben werden muss, **@model** mit kleinem **m** darf nur einmalig zum Zuweisen des Models für die Seite verwendet werden. Was fangen wir nun damit an? Wir öffnen einmal die Datei **index.cshtml.cs**, die wir auch als Code behind-Datei für die Razor Page bezeichnen können. Dort fügen wir eine öffentliche Eigenschaft namens **Seitentitel** wie folgt hinzu:

```
namespace RazorPages.Pages {
    public class indexModel : PageModel {
        public string Seitentitel { get; set; } = "Seitentitel aus dem Model";
        public void OnGet() {
        }
    }
}
```

Diese Eigenschaft können wir nun auch von der Seite **index.cshtml** aus referenzieren und so dafür sorgen, dass der Wert aus der Model-Klasse aus der Code behind-Datei in das Ergebnis, als die **.html**-Seite, eingearbeitet wird. Dies zeigt als Erstes den Seitentitel wie in der **.cshtml**-Datei definiert und dann den Seitentitel aus dem Model an (siehe Bild 7):

```
<h1>@Seitentitel</h1>
<h1>@Model.Seitentitel</h1>
```

Bild 7: Inhalte von verschiedenen Quellen

Inline oder mehrzeilig?

Es gibt zwei Varianten, Razor Code im HTML-Code unterzubringen. Entweder Sie fügen einfache Ausdrücke einfach mit einem führenden

@-Zeichen in den Code ein. Oder Sie verwenden das Konstrukt **@{...}**, um komplette Anweisungen oder auch mehrzeilige Codes zu nutzen. Die einfache Variante haben Sie bereits oben gesehen:

```
<h1>@Seitentitel</h1>
```

Die Variante mit kompletten Anweisungen sieht so aus – hier zunächst mit einer Anweisung:

```
@{ var Seitentitel = "Dies ist der Seitentitel"; }
```

Hier geben wir komplette C#-Anweisungen inklusive Semikolon ein. Sie können aber auch mehrere Anweisungen eingeben:

```
@{
    var Seitentitel = "Dies ist der Seitentitel";
    var AndereInformation = "Andere Information";
}
```

Variablen im Razor-Code

Variablen im Razor-Code werden mit dem Datentyp **var** angegeben. Das unter C# am Zeilenende gängige Semikolon ist auch hier obligatorisch (hier in einer Zeile):

```
@{ var Text = "Dies ist ein Beispieltext."; }
```

Danach können wir im HTML-Code mit **@Text** auf die Variable zugreifen:

```
<p>@Text</p>
```

Wir können den HTML-Code aber interessanterweise auch in den Codeblock integrieren:

```
@{
    var TextIntegriert = "Dies ist ein in den Razor-Code integrierter Beispieltext.";
    <p>@TextIntegriert</p>
}
```

Anführungszeichen oder Backslash in Razor-Code

Wenn Sie innerhalb eines Razor-Blocks Anführungszeichen oder das Backslash-Zeichen nutzen wollen, stellen Sie der in Anführungszeichen angegebenen Zeichenkette ein @-Zeichen voran, also etwa so:

```
@{
    var TextMitSonderzeichen = @"Text mit "" und \"
}
```

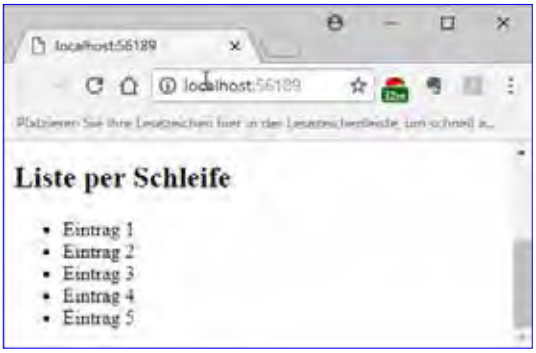


Bild 8: Schleife im **.cshtml**-Code

Webdesign mit Bootstrap Studio

Wer einmal eine Webseite mit CSS designt hat, weiß, wie aufwendig das ist. Selbst das Anpassen bestehender Seiten ist nicht gerade intuitiv, wenn man sonst eher in der Programmierung unterwegs ist. Manchmal führt allerdings kein Weg um Designarbeit herum, und dann ist es wichtig, ein gutes Tool an der Hand zu haben. Die mit ASP.NET Core Razor Pages verwendete Technik namens Bootstrap können Sie beispielsweise mit dem Tool Bootstrap Studio sehr gut in den Griff bekommen. Dieser Artikel zeigt die grundlegenden Schritte zum Designen einer Webanwendung mit Bootstrap Studio.

Das Tool **Bootstrap Studio** finden Sie unter dem Link bootstrapstudio.io. Es kostet regulär 50$, zum Zeitpunkt der Erstellung dieses Artikels gab es einen reduzierten Preis von 25$. Nachdem Sie das Tool heruntergeladen und installiert haben, können Sie es gleich starten. Es liefert dann praktischerweise gleich den **Getting Started Guide** (siehe Bild 1). Bevor wir dort einsteigen, noch ein paar Erläuterungen, warum wir neben Visual Studio noch ein weiteres Tool benötigen. Visual Studio ist eine exzellente Entwicklungsumgebung, aber für die Razor Pages bietet es eben noch keine guten Tools zur Entwicklung der Benutzeroberflächen. Dies ist gerade anspruchsvoll, wenn Sie Webanwendungen bauen wollen, die nicht nur auf einem Endgerät laufen sollen, sondern sich an verschiedene Bildschirmgrößen anpassen. Genau dabei unterstützt uns Bootstrap als ein CSS-Framework, das Vorlagen für die unterschiedlichen Design-Elemente liefert. Die Razor Pages können Bootstrap nutzen, was uns eine Menge Zeit sparen würde, wenn wir die Möglichkeit hätten, das Bootstrap-Framework zur Erstellung der Gestaltungselemente zu nutzen. Und hier setzt Bootstrap Studio an.

Bootstrap Studio

Bootstrap Studio kommt mit drei Bereichen. Der linke liefert, ähnlich wie die Toolbox in Visual Studio, die verschiedenen Elemente, die Sie in Ihre Internetseiten integrieren können. Diese ziehen Sie dann einfach auf den Bereich in der Mitte, der die ak-

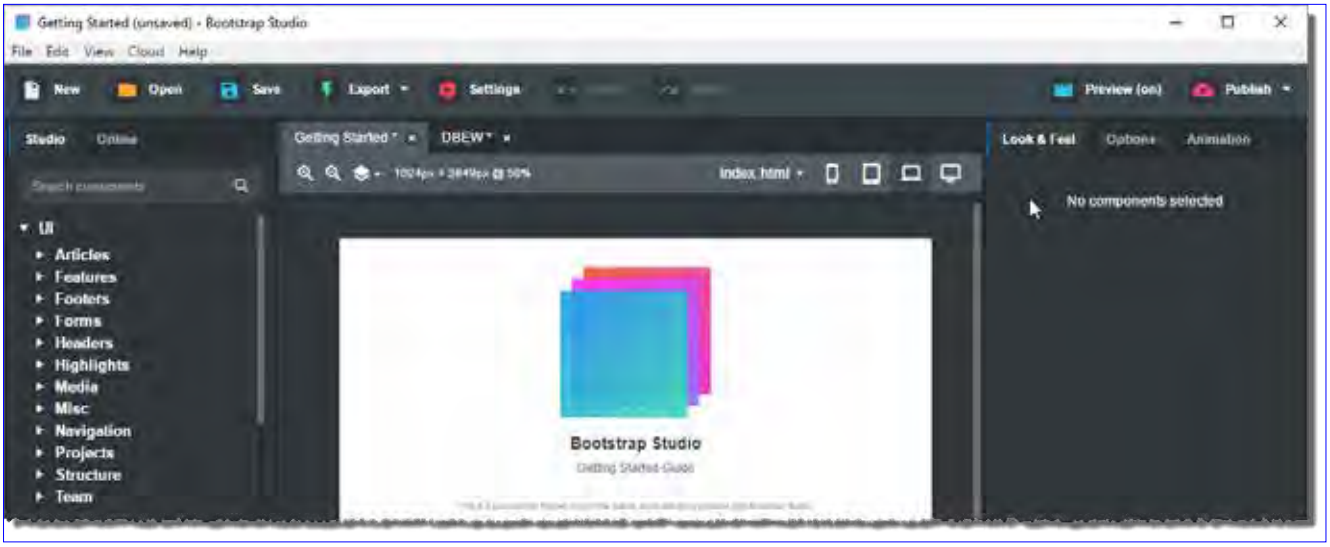


Bild 1: Startbildschirm von Bootstrap Studio

tuelle Ansicht der Internetseite liefert. Sie können hier an beliebigen Stellen Design-Elemente einfügen, wobei Sie zunächst einmal das erste Element hinzufügen müssen. Wenn Sie eine komplette Webseite wie beispielsweise die automatisch beim Anlegen eines neuen ASP.NET Core-Webprojekts generieren wollen, würden Sie beispielsweise wie folgt vorgehen:

- Als Erstes ziehen wir eine geeignete Navigation aus der Toolbox in den Entwurf. Um herauszufinden, welche Navigationen es gibt, klappen Sie den Eintrag **Navigation** auf und überfahren die einzelnen Einträge mit der Maus. Der Eintrag **Navigation with Button** entspricht etwa der Navigation, die Visual Studio anlegt, wenn Sie die Option zum Hinzufügen einer Benutzerverwaltung aktivieren (siehe Bild 2). Ziehen Sie diesen Eintrag einfach in den Entwurf der Seite im mittleren Bereich.

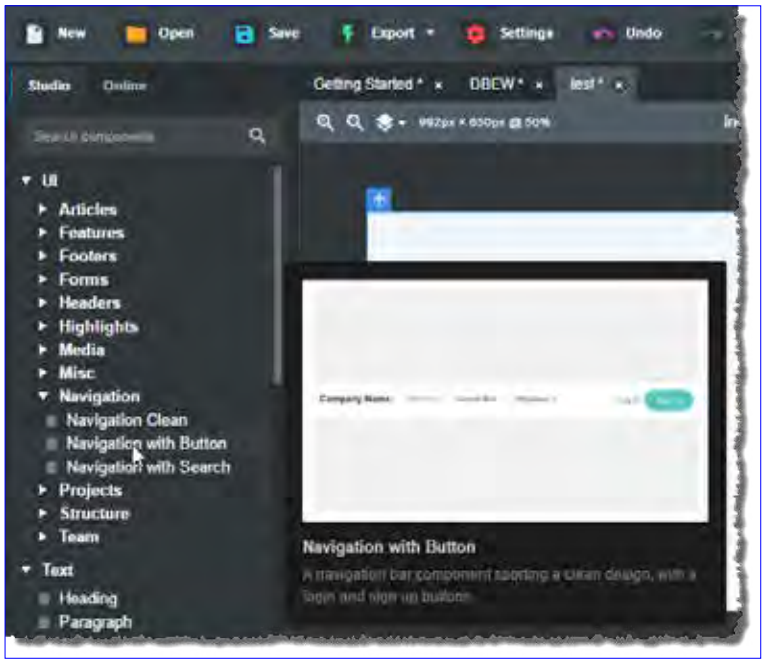


Bild 2: Ansehen der verschiedenen Navigationen

- Das Ergebnis sieht dann wie in Bild 3 aus. Bevor wir dieses Element anpassen, fügen wir zunächst weitere Elemente hinzu.

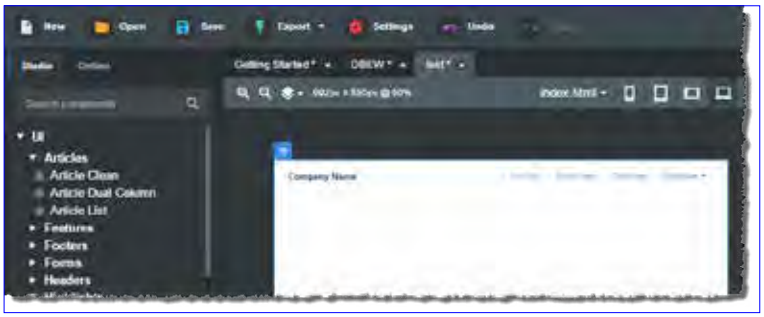


Bild 3: Die Navigation als erstes hinzugefügtes Element

- Danach benötigen wir den Inhalt. Den Slider, den wir im Beispielprojekt von Visual Studio finden, wollen wir an dieser Stelle weglassen und stattdessen eines der **Article**-Templates hinzufügen. Wir überfahren diese wieder, um das für uns passende Template zu finden, und entscheiden uns für die Vorlage **Article Clean**. Diese können wir nun per Drag and Drop an zwei Stellen einsetzen – entweder über oder unter dem bereits vorhandenen Element. Der Artikel soll natürlich darunter platziert werden. Dies fügt auch gleich noch ein paar Bilder hinzu – praktisch, wenn Sie auch Bilder in ihrem Inhalt aufnehmen möchten (siehe Bild 4).

Bild 4: Die Webseite mit Navigation und einem Artikel-Inhaltselement

- Nun fehlt noch der Fußbereich. Diesen finden wir unter dem Element **Footers**. Wir ziehen den Eintrag **Footer Basic** unter dem Inhaltselement in den Entwurf und erhalten

die Ansicht aus Bild 5. Damit haben wir im Prinzip schon den Aufbau des Beispiels nachgebaut, das auch in Visual Studio verwendet wird.

Bild 5: Navigation, Inhalt und Footer – fertig ist die Webseite.

Anpassen des Entwurfs

Nun wollen Sie die Elemente natürlich nicht wie hier zu sehen übernehmen, sondern gegebenenfalls diverse Änderungen vornehmen – beispielsweise am Inhalt, an den Farben von Text und Hintergrund, an der Schriftart und an anderen Elementen der Benutzeroberfläche. Vielleicht möchten Sie auch noch Elemente entfernen oder hinzufügen.

Navigation bearbeiten

Die Navigation besteht aktuell aus dem Namen der Seite, einigen Navigationslinks und einem Dropdown-Menü. Wenn Sie zum Beispiel den Namen der Seite anpassen möchten, klicken Sie auf das Element, klicken auf das Symbol zum Bearbeiten und ändern dann den Inhalt des Elements. Auf die gleiche Art passen Sie die Beschriftungen der Navigationslinks an. Die Bearbeitung der Beschriftungen können Sie allerdings auch ganz einfach per Doppelklick auf das jeweilige Element aktivieren (siehe Bild 6).

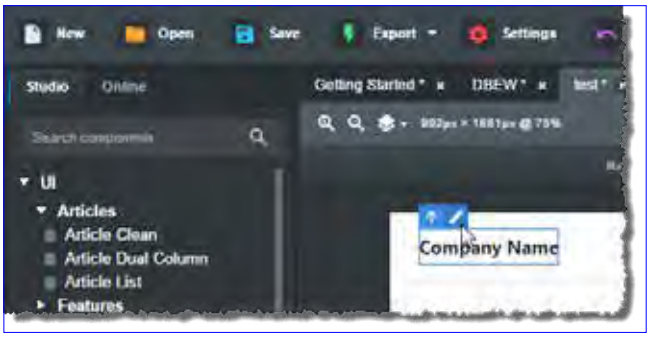


Bild 6: Anpassen der Beschriftung

Wenn Sie Navigationseinträge hinzufügen oder entfernen oder als aktiv oder inaktiv markieren möchten, erledigen Sie das über die Schaltflächen über dem Entwurf (siehe Bild 7). Im Bild haben wir etwa durch Anklicken der Schaltfläche **After** ein neues Element namens **Nav Item** rechts vom Element **First Item** hinzugefügt.

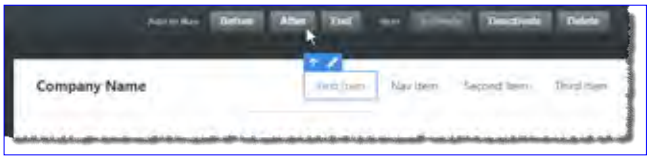


Bild 7: Bearbeiten der Navigationsleiste

Übergeordnetes Element markieren

Manchmal klicken Sie ein Element an, um es beispielsweise zu löschen, und der dann um das Element gezeichnete Rahmen bietet nur die beiden Einträge mit dem **Pfeil nach oben**- und dem **Bearbeiten**-Eintrag an. Der Grund, warum Sie das Element zu diesem Zeitpunkt nicht löschen können, ist, dass aktuell das Beschriftungs-Element markiert ist, das zu dem Objekt gehört. Zum Löschen klicken Sie einmal auf die **Select Parent**-Schaltfläche mit dem Pfeil nach oben, dann finden Sie in der Regel auch die Befehle zur weiteren Bearbeitung wie etwa die **Löschen**-Schaltfläche.

Navigationselemente bearbeiten

Wenn Sie nun die Navigationselemente bearbeiten wollen, können Sie neue Elemente an der gewünschten Stelle hinzufügen, die vorhandenen Elemente verschieben, die Beschriftungen anpassen oder Elemente löschen.

Wenn Sie das Dropdown in der Navigationsleiste bearbeiten wollen, klicken Sie dieses zunächst an. Dann finden Sie oben links neben der Beschriftung **Dropdown** eine Schaltfläche namens **Open**. Damit klappen Sie das Dropdown-Element auf, um seine Einträge zu bearbeiten.

Stellt sich noch die Frage, wie Sie ein weiteres Dropdown zur Navigationsleiste hinzufügen. Wie Sie einfache Button-Elemente zur Navigationsleiste hinzufügen, haben Sie ja schon erfahren. Und auch das Hinzufügen von weiteren Dropdowns gelingt ganz einfach: Diese finden Sie nämlich in der Toolbox unter dem Eintrag **Controls**. Das Dropdown-Element ziehen Sie dann einfach von dort an die gewünschte Stelle in der Navigationsleiste.

Bild 8: Hinzufügen einer Bilddatei

Inhalt bearbeiten

Den Inhalt können Sie zunächst durch Eingabe der gewünschten Texte bearbeiten. Wenn Sie Texte in verschiedenen Formatierungen benötigen, können Sie jeweils neue Absätze aus dem Bereich **Text** der Toolbox an die gewünschte Stelle ziehen – also etwa Überschriften (**Heading**) oder einfache Absätze (**Paragraph**).

Sie können auch beispielsweise neue Bilder zum Inhalt hinzufügen. Dazu ziehen Sie das Element **ImageIImage** an die gewünschte Stelle im Inhalt. Gleich nach dem Hinzufügen erscheint zunächst ein viereckiger Platzhalter. Per Doppelklick auf diesen öffnen Sie den Dialog **Choose an Image**, der die Möglichkeit bietet, bereits vorhandene Bilder einzufügen oder über die Schaltfläche **Import Image** neue Bilder zur Vorlage hinzuzufügen (siehe Bild 8). Klicken Sie dann eines der angezeigten Bilder an, erscheint dieses im Inhalt.

Fußzeile bearbeiten

Wenn Sie zum Beispiel keine Links auf Twitter, Facebook und Co. im Footer anzeigen möchten, können Sie diese einfach entfernen. Die dort angezeigten Elemente bearbeiten Sie grundsätzlich genauso wie die in der Navigationsleiste. Auch den Namen des Unternehmens ganz unten passen Sie beispielsweise per Doppelklick und Ändern des Inhalts an.

Mit Rastern arbeiten

Wenn Sie kein schnödes einzeiliges Layout wünschen, können Sie über die Grid-Elemente auch ein Raster definieren, das Ihre Inhalte anzeigt. Dazu ziehen Sie zunächst ein **Row**-Element aus dem Bereich **Grid** der Toolbox an die gewünschte Stelle. Diese trägt dann die Beschriftung **Empty Row** (siehe Bild 9).

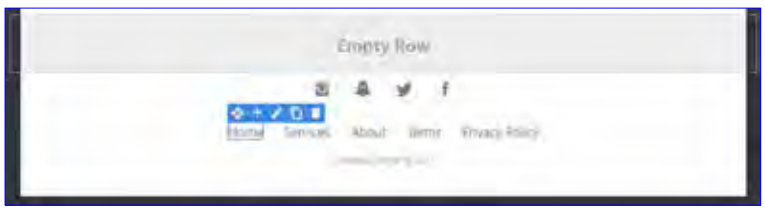


Bild 9: Hinzufügen eines **Row**-Elements als Basis für ein Grid

Dem neuen Element können Sie nun **Column**-Elemente hinzufügen. Ziehen Sie eines aus der Toolbox in ein **Empty Row**-Element, wird dieses mit dem **Empty Column**-Element gefüllt. Diesem können Sie nun ein weiteres **Column**-Element hinzufügen. Um festzulegen, ob diese links oder rechts vom bereits vorhandenen **Column**-Element eingefügt wird, lassen Sie es erst dann fallen, wenn links oder rechts nach dem Überfahren mit der Maus ein entsprechender Pfeil auftaucht. Experimentieren Sie einfach damit herum. Sie können damit dann beispielsweise solche Konstrukte wie in Bild 10 erzeugen.

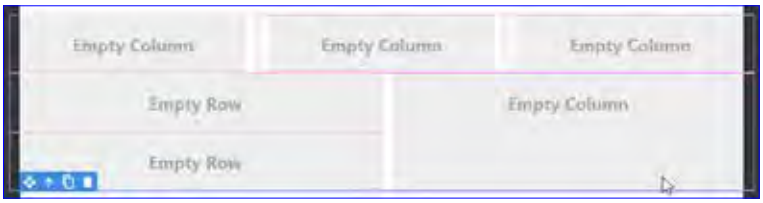


Bild 10: Zeilen und Spalten als Grid

Überführen des Entwurfs in Razor Pages

Nun wird es interessant: Wir haben jetzt eine Seite mit Seitennavigation, Inhalt und Fußzeile entworfen und eventuell bereits mit Inhalten gefüllt. Nun wollen wir die so erzeugten Inhalte in eine frisch erzeugte Webanwendung in Visual Studio auf Basis der Vorlage **ASP.NET Core Webanwendung** mit dem Typ **Webanwendung** überführen.

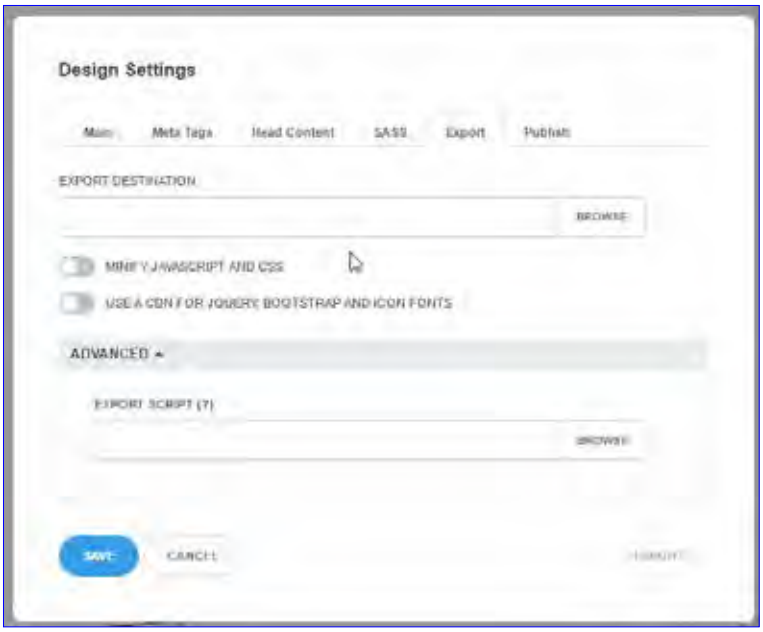


Bild 11: Export-Einstellungen

Dazu benötigen wir zuerst einmal den Code für die Elemente des frisch erzeugten Entwurfs. Bootstrap Studio bietet dazu verschiedene Export-Funktionen. Diese starten wir über den Menüeintrag **Export**. Es erscheint der Dialog aus Bild 11, dessen Einstellungen wir übernehmen können. Wichtig ist die Schaltfläche **Save**, mit der wir die Einstellungen speichern können. Mit der Schaltfläche **Export** exportieren Sie schließlich die Webseite – aber erst, nachdem Sie auf der Registerseite **Export** das Zielverzeichnis ausgewählt haben. Dies exportiert dann den HTML-Code in eine Datei namens **index.html** und die übrigen Elemente wie die **.css**-Dateien et cetera einen Ordner namens **assets**.

Was machen wir nun damit? Wie wir bereits im Beitrag **ASP.NET Core-Anwendung anpassen** beschrieben haben, teilen sich die eigentlich HTML-Daten auf die beiden Dateien **_Layout.cshtml** und in die jeweilige Datei mit dem anzuzeigenden Inhalt auf, also beispielsweise **Index.cshtml** oder **About.cshtml**. Genau auf diese Dateien müssen wir nun auch den in die Datei **index.html** exportieren HTML-Code aufteilen.

Dabei können wir den Inhalt nicht einfach aufteilen und in die beiden Dateien **_Layout.cshtml** und **Index.cshtml** schreiben, sondern wir müssen zuvor noch schauen, welche dynamischen Elemente – sprich Markup – sich in der aktuellen Version dieser Dateien befindet. So enthält die Datei **_Layout.cshtml** ja beispielsweise ein Element, für das später der Inhalt von **Index.cshtml** eingebettet wird.

E-Mails per Sendgrid verschicken

Wenn Sie am Desktop arbeiten, können Sie E-Mails einfach per Outlook verschicken. Das gelingt auch ferngesteuert etwa von einer Desktop-Anwendung auf Basis von .NET aus. Wenn Sie jedoch Mails sicher über eine Webanwendung verschicken wollen, wird es interessant. Sollten Sie keinen eigenen Mailserver auf dem Internetserver betreiben, können Sie auf einen Dienstleister ausweichen, der Ihnen eine entsprechende Schnittstelle anbietet. Microsoft empfiehlt hier den Anbieter Sendgrid. Wir zeigen, wie Sie sich registrieren und für den ASP-gesteuerten Versand von E-Mails vorbereiten. Der Clou: Bis zu einer bestimmten Menge von E-Mails ist der Service kostenlos.

Sendgrid

Sendgrid ist ein US-amerikanischer Anbieter, der erstmal eines für Sie erledigt: das Versenden von E-Mails. Dafür gibt es verschiedene Pakete, die sowohl die Menge der zu versendenden E-Mails als auch die Anzahl der zu speichernden Kontakte festlegen (siehe Bild 1). Das günstigste Paket heißt **Free**, was sich auf den monatlichen Preis bezieht – es ist nämlich kostenlos. Je mehr E-Mails verschickt und Adressen verwaltet werden sollen, desto höher sind die monatlichen Kosten. Für Testzwecke reicht uns jedoch das Paket **Free** völlig aus – hier können Sie 40.000 E-Mails in den ersten 30 Tagen und danach täglich 100 E-Mails verwenden.

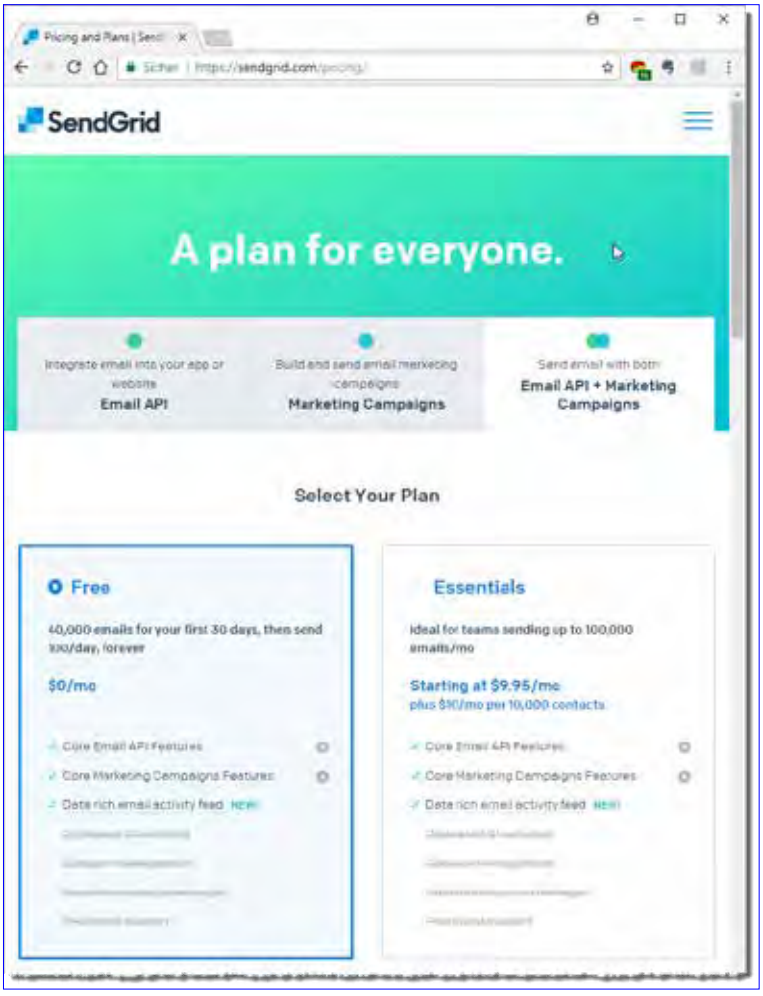


Bild 1: Der E-Mail-Versender **Sendgrid**

Für eine Webseite, die ein paar Bestätigungsmails und ein paar Kontaktanfragen versendet, reicht das locker aus. Auch das nächstgrößere Paket, das bereits den Versand von 100.000 E-Mails pro Monat erlaubt, kostet nur 9,95 $ pro Monat.

E-Mails und mehr

Neben dem über verschiedene Schnittstellen steuerbaren Versand von E-Mail liefert der Anbieter noch weitere Dienstleistungen – so können Sie etwa Newsletter darüber versenden und die dazu notwendigen Kontakte verwalten. Dabei sind typische Funktionen wie etwa das Bereitstellen eines Abmeldelink für den Newsletter und automatisches Austragen aus der Empfängerliste bereits implementiert. Wer etwa eine Newsletter-Anmeldung per Double-Opt-In realisieren möchte, muss allerdings eine eigene Funktion dafür implementieren.

Kostenloses Paket

Wir wählen für den Beginn das Paket **Free** aus. Aktuell wollen wir diesen Dienstleister dafür nutzen, die in dem Artikel **Authentifizierung unter ASP.NET Core** benötigten Bestätigungsmails zu versenden.

Nach der Auswahl des gewünschten Pakets und dem Anklicken des Links **Try for free** landen Sie auf der Registrierungsseite. Hier geben Sie den Benutzernamen, das Kennwort und Ihre E-Mail-Adresse an. Nach dem Bestätigen der obligatorischen Optionen geht es dann mit einem Klick auf **Create Account** weiter (siehe Bild 2). Im Bereich rechts können Sie die aktuell anfallenden Kosten einsehen.

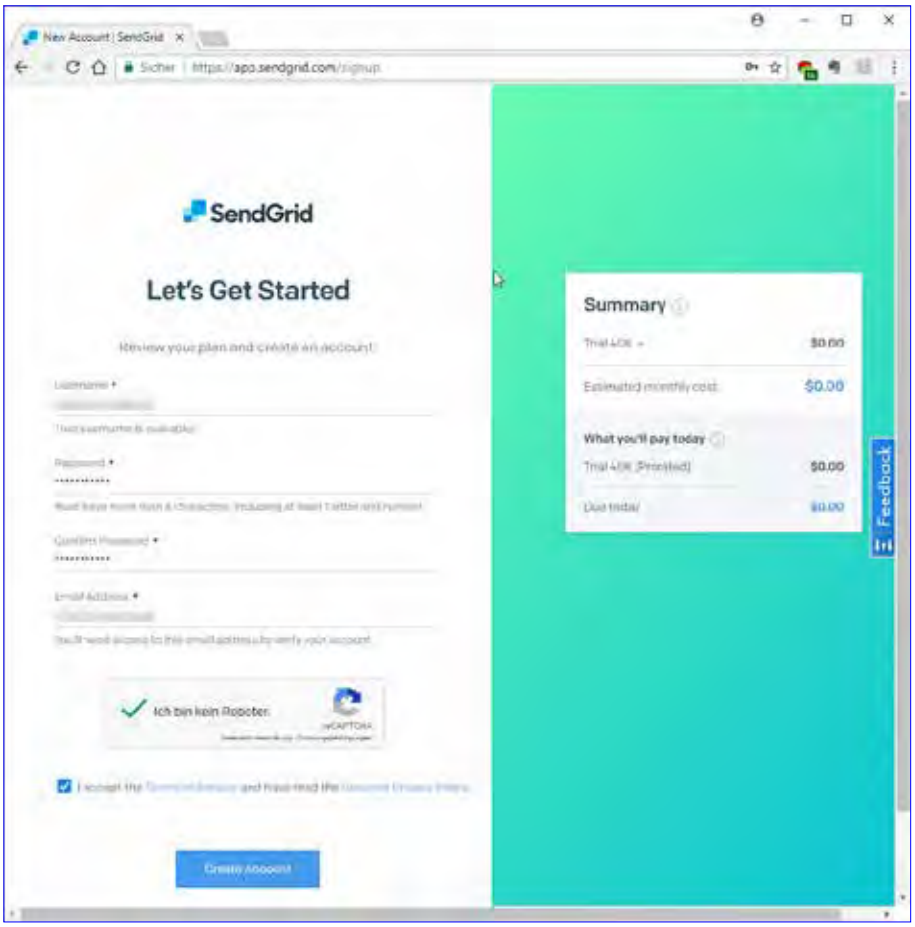


Bild 2: Beispielfenster zum Versenden von E-Mails

Die folgende Seite fragt dann noch einige persönliche Daten ab, bevor es nach einem Klick auf **Get Startet!** losgeht. Auf der Willkommen-Seite können wir nun auswählen, auf welche Weise wir mit Sendgrid arbeiten wollen. Wir möchten Mails per Web API verschicken, also wählen wir den passenden **Start**-Button (siehe Bild 3).

Der nächste Dialog fragt nach einer Setup-Methode. Hier haben Sie die Auswahl zwischen **Web API** und **SMTP Relay**. Wir wählen hier den Eintrag **Web API**, da wir den Versand per C#

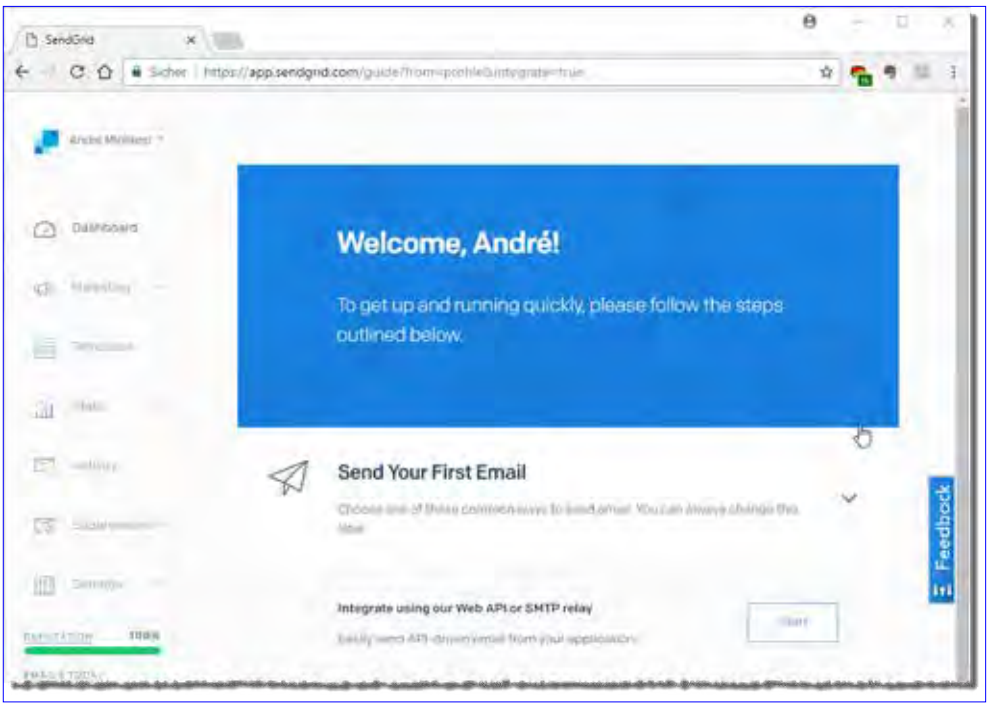


Bild 3: Auswahl des benötigten Services

programmieren wollen. Noch eine Seite später wählen Sie die Sprache aus, mit der Sie auf die Web API zugreifen wollen. Hier fällt die Wahl auf **C#**.

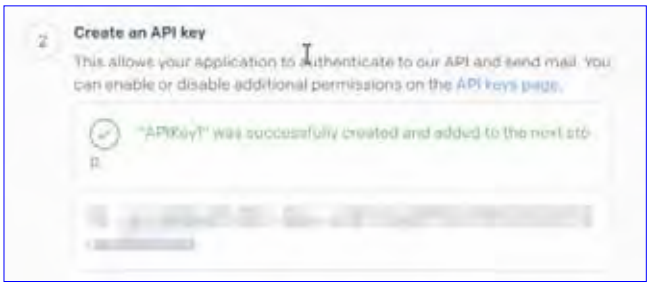


Bild 4: Erfolgreich angelegter API-Key

Danach müssen wir wieder arbeiten: Als Erstes prüfen wir die Voraussetzungen, die hier mit .NET 4.5.2 angegeben sind. Diese müssen Sie erfüllen. Als Zweites legen wir einen API-Key an, der einen sicheren Zugriff von der Web-Anwendung auf die Schnittstelle von Sendgrid erlaubt. Dazu geben Sie unter **Create an API key** einen Namen für den Key ein, beispielsweise **APIKey1**, und klicken auf **Create Key**. Dann erhalten Sie den Key wie in Bild 4. Den dritten Schritt mit dem Titel **Create an environment variable** überspringen wir zunächst.

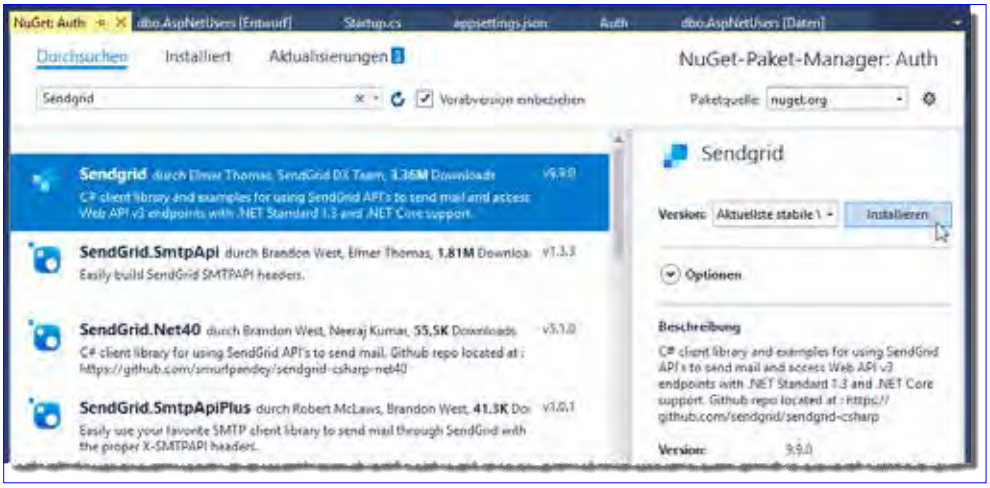


Bild 5: Installieren des NuGet-Pakets **Sendgrid**

Projekt anlegen und fortfahren

Dafür fahren wir mit dem vierten Schritt fort, für den Sie zunächst ein neues Projekt erstellen. Zu Beispielzwecken legen wir eine einfache C#-Konsolenanwendung an. Bevor wir die Anwendung nun mit dem Beispielcode versehen, müssen wir noch das Sendgrid-Paket hinzufügen. Dann folgt der vierte Schritt auf der Sendgrid-Webseite: **Install the package**. Ziel dieses Schritts ist die Installation des NuGet-Pakets für den Zugriff auf **Sendgrid**. Dazu starten Sie den NuGet-Dialog über den Kontextmenü-Eintrag **NuGet-Pakete verwalten...** des Projekts im Projektmappen-Explorer. Hier wechseln Sie zum Bereich **Durchsuchen** und suchen nach dem Eintrag **Sendgrid**. Diesen markieren Sie dann und klicken im rechten Bereich auf die Schaltfläche **Installieren** (siehe Bild 5).

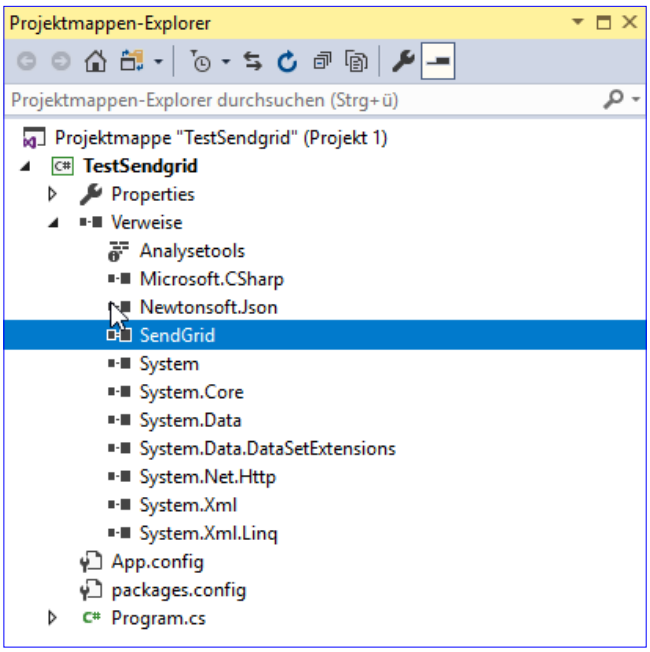


Bild 6: Die Sendgrid-Erweiterung in den Verweisen

Die Sendgrid-Bibliothek finden Sie dann beispielsweise im Projektmappen-Explorer unter **<Projektname>|Abhängigkeiten|NuGet** (siehe Bild 6).

Test-E-Mail versenden

Um eine Test-E-Mail zu versenden, fügen wir der Klasse **Program** den folgenden Code hinzu. Für die darin verwenden Objekttypen benötigen wir allerdings noch ein paar using-Anweisungen, die wir im Kopf der Klassendatei unterbringen:

```
using System.Threading.Tasks;
using Sendgrid;
using Sendgrid.Helpers.Mail;
```

Die Methode **Main** ruft dabei den asynchronen Task **SendMail** auf. Das Erstellen einer E-Mail würden wir normalerweise eher eine ASP.NET-Anwendung zuschreiben, daher programmieren wir die Methode, welche die E-Mail zusammenstellt und versendet als asynchronen Task – und es gibt noch einen weiteren Grund dafür, den wir weiter unten erläutern. Die asynchrone Methode erkennen Sie am Schlüsselwort **async** und am Methodentyp **Task**. Damit der Aufruf dennoch wartet, bis die asynchrone Methode durchgeführt wurde, rufen wir diese mit der **Wait**-Methode auf:

```
class Program {
    private static void Main() {
        SendMail().Wait();
    }
    static async Task SendMail() {
```

Den soeben ermittelten API-Key schreiben wir in die **string**-Variable **apiKey**. Normalerweise würden wir diesen an anderer Stelle speichern, wo er auch zur Laufzeit geändert werden kann, aber für Testzwecke wählen wir die einfache Möglichkeit:

```
        string apiKey = "SG.HT7pvIFvQJWK12BMBcoEq-yg.xXWtoVKw3NlpYBld7nrVY3429xKomFUL1ZUcGHe2n0";
```

Dann erstellen wir ein neues Objekt des Typs **SendgridClient** und übergeben dieser beim Erstellen den API-Key aus **apiKey**:

```
        SendgridClient SendgridClient = new SendgridClient(apiKey);
```

Dann legen wir ein neues Objekt des Typs **EmailAddress** namens **from** an und weisen diesem die E-Mail-Adresse und den Namen des Absenders zu:

```
        EmailAddress from = new EmailAddress("andre@minhorst.com", "André Minhorst");
```

Die Variablen **subject** nimmt den Betreff auf:

```
        string subject = "Beispiel per Console-App";
```

Die Empfänger-Adresse landet in der Variablen **to**:

```
        EmailAddress to = new EmailAddress("andre@minhorst.com", "André Minhorst");
```

Razor Pages: Von Seite zu Seite

Wenn Sie eine Webanwendung zur Verwaltung von Daten programmieren, wollen Sie Daten anzeigen, eingeben, löschen und so weiter. Das gelingt nicht ohne Navigation zwischen den verschiedenen Seiten. Die Hauptseiten sind über die Navigationsleisten erreichbar, aber wenn Sie etwa einen bestimmten Datensatz anzeigen, Daten von einer Seite zur nächsten übermitteln oder nach dem Löschen eines Datensatzes wieder die Übersicht einblenden wollen, benötigen Sie entsprechende Techniken. Dieser Artikel zeigt, wie Sie mit Links, Senden von Formularen und anderen Methoden von Seite zu Seite gelangen und die gewünschten Daten übermitteln. Dabei spielen auch die Ereignis-Handler einer Seite eine Rolle.

Einfache Links

Einen einfachen Link erstellen Sie in herkömmlichen HTML-Seiten mit dem **a**-Element. In Razor Pages verwenden Sie ein anderes Attribut, um die Zielseite anzugeben – nämlich **asp-page**. Wenn Sie etwa von der **Index.cshtml**-Seite im Verzeichnis **Pages** des Projekts auf die **About**-Seite verlinken wollen, können Sie das etwa mit dem folgenden Code erledigen:

Bild 1: Einfacher Link

```
@page
@model IndexModel
@{
    ViewData["Title"] = "Home page";
}
<h2>Link-Beispiele</h2>
<p><a asp-page="About">Zur Seite 'About'</a></p>
```

Das Ergebnis sieht dann etwa wie in Bild 1 aus. Da sich die Seite **About.cshtml** im gleichen Verzeichnis wie die Seite **Index.cshtml** befindet, reicht die Angabe des Namens der Seite ohne die Dateiendung **.cshtml**. Ein Klick auf diesen Link zeigt dann die Seite **About.cshtml** im Browser an.

Handler-Methoden

Unter ASP.NET Core/Razor Pages gibt es verschiedene Handler-Methoden. Diese heißen beispielsweise wie folgt:

- **OnPost**

- **OnGet**
- **OnPostAsync**
- **OnGetAsync**

Die Code behind-Datei einer Razor Page enthält normalerweise die folgende Handler-Methode:

```
public void OnGet() {

}
```

Diese wird gleich beim Laden der Seite auf der Serverseite ausgelöst. Dies können wir prüfen, indem wir der Methode einen Haltepunkt hinzufügen und das Projekt starten. Bevor die Seite im Browser erscheint, hält die Methode an der Stelle mit dem Haltepunkt an. Auch wenn Sie die Seite mit der Taste **F5** aktualisieren, wird die Methode wieder ausgelöst.

In der durch unsere Seite **Index.cshtml** verlinkten Seite **About.cshtml** haben wir die gleiche Prozedur angelegt und mit einem Haltepunkt versehen. Klicken wir zur Laufzeit auf den Link, löst dies die Methode **OnGet** dieser Seite aus.

Methode OnPost auslösen

Nun wollen wir eine zweite Handler-Methode namens **OnPost** hinzufügen, die genau gleich aufgebaut ist wie **OnGet**. Auch diese fügen wir wieder beiden Pages hinzu, also **Index.cshtml** und **About.cshtml**:

```
public void OnPost() {

}
```

Um diese zu testen, benötigen wir auf der Seite **Index.cshtml** ein Formular. Dieses erhält die Methode **post** für das Attribut **method** und eine Schaltfläche zum Absenden des Formulars:

```
<form method="post">
    <button class="btn btn-default">Zur gleichen Seite.</button>
</form>
```

Wenn wir nun auf diese Schaltfläche klicken, wird die Methode **OnPost** der aufgerufenen Seite **Index.cshtml** aufgerufen – diese wird dann auch erneut angezeigt.

Nun fügen wir eine weitere Schaltfläche in einem neuen Formular hinzu, welche die Seite **About.cshtml** aufrufen soll. Standardmäßig wird beim Absenden eines Formulars immer die Seite aufgerufen, auf der sich das Formular befindet. Um eine andere Seite wie etwa **About.cshtml** aufzurufen, fügen Sie dem **button**-Element das Attribut **asp-page** mit dem Namen der aufzurufenden Datei hinzu:

```
<form method="post">
    <button class="btn btn-default" asp-page="About">Zur Seite 'About'.</button>
</form>
```

Dies öffnet die Seite **About.cshtml** und führt dort die Handler-Methode **OnPost** aus.

Mehrere Formulare für die gleiche Seite

Im obigen Fall haben wir nun zwei Formulare, die allerdings verschiedene Seiten aufrufen. Wenn Sie jedoch verschiedene Formulare auf einer Seite nutzen, die alle die gleiche Seite aufrufen – beispielsweise die aktuelle Seite – gelingt dies natürlich. Allerdings wird immer die gleiche Handler-Methode aufgerufen, standardmäßig **OnPost**. Wenn Sie mehrere Formulare nutzen wollen, die zur Ausführung verschiedener **OnPost**-Methoden führen sollen, legen Sie etwa die folgenden Formulare an:

```
<h3>Zwei Formulare zu verschiedenen OnPost-Methoden</h3>
<form method="post" asp-page-handler="Eins">
    <button class="btn btn-default">Zur Methode OnPostEins.</button>
</form>
<form method="post" asp-page-handler="Zwei">
    <button class="btn btn-default" >Zur Methode OnPostZwei.</button>
</form>
```

Die Seite sieht mittlerweile wie in Bild 2 aus. Wenn Sie nun nacheinander auf die beiden Schaltflächen **Zur Methode OnPostEins** und **Zur Methode OnPostZwei** anklicken, wird nun beide Male die Methode **OnPost** des Formulars **Index.cshtml** ausgeführt. Es hat sich also erstmal nichts geändert.

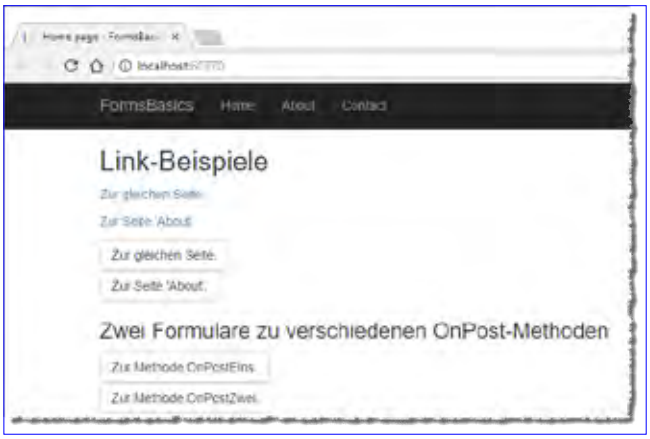


Bild 2: Beispiel-Links und -Formulare

Um tatsächlich zwei verschiedene Methoden aufzurufen, müssen Sie diese nach einem bestimmten Schema anlegen. Dieses besteht aus dem eigentlichen Namen der Handler-Methode, also **OnPost**, und dem mit **asp-page-handler** angegebenen Ausdruck. Die beiden von den beiden Formularen adressierten Handler-Methoden heißen also **OnPostEins** und **OnPostZwei** und sehen wie folgt aus:

```
public void OnPostEins() {

}

public void OnPostZwei() {

}
```

Beim Anklicken der beiden Schaltflächen werden nun diese beiden Handler-Methoden aufgerufen. Diese Technik können Sie also beispielsweise nutzen, wenn Sie die angezeigten Daten einer Datenbank bearbeiten, löschen oder anzeigen wollen.

Wie wird der **Page**-Handler nun übermittelt? Wie Bild 3 zeigt, über die Parameterliste beim Aufruf der Seite.

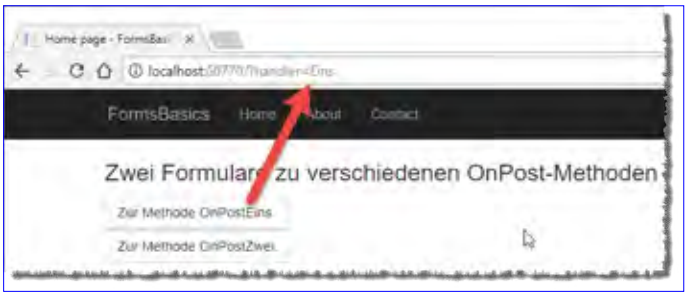


Bild 3: **Page**-Handler als Parameter beim Seitenaufruf

Ein Formular mit mehreren Schaltflächen

Sie können auch zwei verschiedene Aktionen mit zwei Schaltflächen in einem **form**-Element unterbringen. Dazu müssen Sie lediglich das Attribut **asp-page-handler** vom **form**-Element in die **button**-Elemente übertragen.

Folgendes Beispiel funktioniert genauso wie das vorherige:

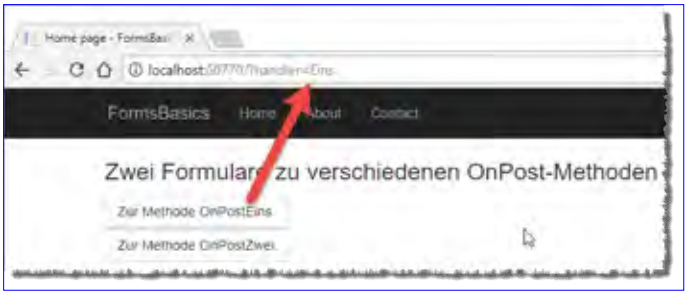


Bild 4: Kein Parameter mit **Page**-Handler bei der **Get**-Methode

```
<h3>Zwei Schaltflächen eines Formulars zu verschiedenen OnPost-Methoden</h3>
<form method="post">
    <button class="btn btn-default" asp-page-handler="Eins">Zur Methode OnPostEins.</button>
    <button class="btn btn-default" asp-page-handler="Zwei">Zur Methode OnPostZwei.</button>
</form>
```

Formulare mit Get

Wir sehen also, dass bei Formularen mit der Post-Methode die Parameter – wie hier zum Beispiel der Wert des **Page**-Handlers – per Parameter an die URL angehängt werden. In manchen Fällen macht es Sinn, einen alternativen Ansatz zu wählen, nämlich die **Get**-Methode. Der einzige Unterschied zur **Post**-Methode im HTML-Code ist, dass Sie statt dem Wert **post** den Wert **get** für das Attribut **method** des Formulars festlegen. Dazu legen wir zwei weitere Formulare an:

```
<h3>Zwei Formulare zu verschiedenen OnGet-Methoden</h3>
<form method="get" asp-page-handler="Eins">
    <button class="btn btn-default">Zur Methode OnGetEins.</button>
</form>
<form method="get" asp-page-handler="Zwei">
    <button class="btn btn-default">Zur Methode OnGetZwei.</button>
</form>
```

Starten Sie nun die Anwendung und klicken diese beiden Schaltflächen an, lösen Sie jeweils die Methode **OnGet** der Code behind-Klasse der Seite **Index.cshtml** aus. Allerdings sehen wir auch, dass der Page-Handler hier nicht als Parameter übergeben wird (siehe Bild 4). Legen wir also nun die beiden passenden Handler-Methoden namens **OnGetEins** und **OnGetZwei** an:

```
public void OnGetEins() {

}
public void OnGetZwei() {

}
```

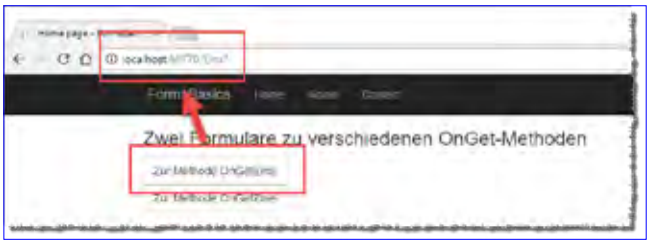


Bild 5: Auslösen einer Get-Handler-Methode

Normalerweise würde man annehmen, dass nun die beiden Methoden **OnGetEins** und **OnGetZwei** ausgelöst werden, wenn der Benutzer auf eine der beiden neuen Schaltflächen klickt. Dies ist allerdings nicht der Fall: Das Anklicken der Schaltflächen löst nach wie vor die Handler-Methode **OnGet** aus. Damit dies dennoch funktioniert, müssen wir auf der Seite **Index.cshtml** noch die Direktive **@page** anpassen:

```
@page "{Handler?}"
```

Wenn Sie nun etwa auf die Schaltfläche des Formulars mit **method="get"** und **asp-page-handler="Eins"** klicken, wird erstens die Methode **OnGetEins** aufgerufen. Zweitens wird der Wert des Attributs **asp-page-handler** in die URL integriert (siehe Bild 5).

Parameter für Handler-Methoden festlegen und verarbeiten

Wenn Sie eine Handler-Methode aufrufen, wollen Sie gegebenenfalls Informationen übergeben – wie beispielsweise die ID eines zu bearbeitenden oder zu öffnenden Datensatzes. In diesem Fall legen wir beispielsweise drei **form**-Elemente an, welche alle drei jeweils eine Schaltfläche zum Aufrufen des fiktiven Datensatzes enthalten und zusätzlich ein **input**-Element, das durch den Wert **hidden** für das Attribut **type** ausgeblendet erscheint und den Namen und den Wert des zu übermittelnden Parameters enthält:

```
<form method="post" asp-page-handler="Anzeigen">
    <button class="btn btn-default">Datensatz mit ID = 1 anzeigen</button>
    <input type="hidden" name="id" value="1" />
</form>
<form method="post" asp-page-handler="Anzeigen">
    <button class="btn btn-default">Datensatz mit ID = 2 anzeigen</button>
    <input type="hidden" name="id" value="2" />
</form>
<form method="post" asp-page-handler="Anzeigen">
    <button class="btn btn-default">Datensatz mit ID = 3 anzeigen</button>
    <input type="hidden" name="id" value="3" />
</form>
```

Diesen Elementen hängen wir schließlich noch einen als Überschrift 3 formatierten Absatz an, der die Eigenschaft **Meldung** von **@Model** liefert. **@Model** ist ein Platzhalter für die unter **@model** ganz oben angegebene Klasse. Damit können Sie auf öffentliche Eigenschaften der Code behind-Klasse zugreifen:

Kontaktformular unter ASP.NET Core

Wer Anwendungen für den Desktop entwickelt hat, kennt die Vorgehensweise: Bei einem Klick auf OK werden die eingegebenen Werte in einer Ereignisprozedur ausgewertet, gespeichert oder anderen Schritten unterzogen. Bei Internetanwendungen ist das aufgrund der Eigenarten der Browser etwas anders. Hier klicken Sie zwar nach der Eingabe von Daten auch auf eine Schaltfläche, um die Daten zu verarbeiten, aber es gibt dort keine Ereignisprozeduren im klassischen Sinne. Wie die Datenverarbeitung hier geschieht, zeigt der vorliegende Artikel.

Für die Beispiele dieses Artikels legen Sie wieder ein Projekt auf Basis der Vorlage **Visual C#|Web|ASP.NET Core-Webanwendung** an. Wählen Sie im Dialog **Neue ASP.NET Core Webanwendung** den Eintrag **Webanwendung** aus.

Wir wollen zunächst nur eine Seite namens **Index.cshtml** nutzen, die wir allerdings mit eigenen Inhalten füllen wollen. Also entfernen wir den aktuell in der Datei **/Pages/Index.cshtml** enthaltenen Code mit Ausnahme der Razor-Anweisungen zu Beginn des Codes (siehe Bild 1).

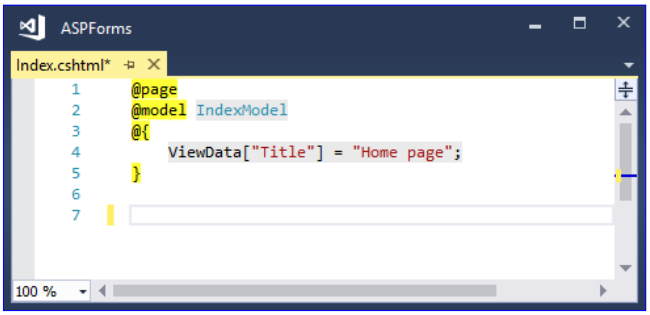


Bild 1: Basis unseres Formulars

In unserem Beispiel wollen wir ein Formular erstellen, in das der Benutzer die Daten wie in einem Kontaktformular eingibt und dessen Inhalt dann an den Betreiber der Webseite geschickt wird. Als E-Mail-Provider wollen wir dann wie im Artikel **E-Mails per SendGrid verschicken** den Anbieter **Sendgrid** nutzen.

Model anlegen

Bevor wir mit der Programmierung des eigentlichen Formulars im HTML-Format beginnen, stellen wir ein Model zusammen, oder genauer: eine Klasse mit einem Model für die E-Mail-Daten. Diese Klasse soll wie folgt aussehen und kann der Einfachheit halber direkt in der Datei **Index.cshtml.cs** eingefügt werden:

```
public class KontaktformularModel {
    [Required]
    public string Vorname { get; set; }
    [Required]
    public string Nachname { get; set; }
    [Required]
    public string Email { get; set; }
    [Required]
    public string Nachricht { get; set; }
}
```

Hier sehen Sie neben den üblichen Elementen einer Klasse jeweils das Attribut **[Required]**. Damit dieses nicht als Fehler markiert wird, fügen wir der Klassendatei mit der **using**-Anweisung den Namespace **System.ComponentModel.DataAnnotations** hinzu:

```
using System.ComponentModel.DataAnnotations;
```

Damit wir ein Objekt auf Basis der Klasse **KontaktformularModel** im HTML-Formular nutzen können, machen wir dieses ähnlich wie unter WPF öffentlich verfügbar (siehe Bild 2):

```
[BindProperty]
public KontaktformularModel Kontakt{ get; set; }
```

Damit kommen wir nun zum HTML-Code des Formulars. Die einfachste Form dieses Formulars sieht wie folgt aus – wenn wir das Projekt nun starten, erhalten wir das Formular aus Bild 3:

```
<form method="post">
    <table>
        <tr>
            <td>Vorname:</td>
            <td><input asp-for="Kontakt.Vorname" /></td>
        </tr>
        <tr>
            <td>Nachname:</td>
            <td><input asp-for="Kontakt.Nachname" /></td>
        </tr>
        <tr>
            <td>E-Mail:</td>
            <td><input asp-for="Kontakt.Email" /></td>
        </tr>
        <tr>
            <td>Nachricht:</td>
            <td><input asp-for="Kontakt.Nachricht" /></td>
        </tr>
        <tr>
            <td colspan="2"><button type="submit">Absenden</button></td>
        </tr>
```

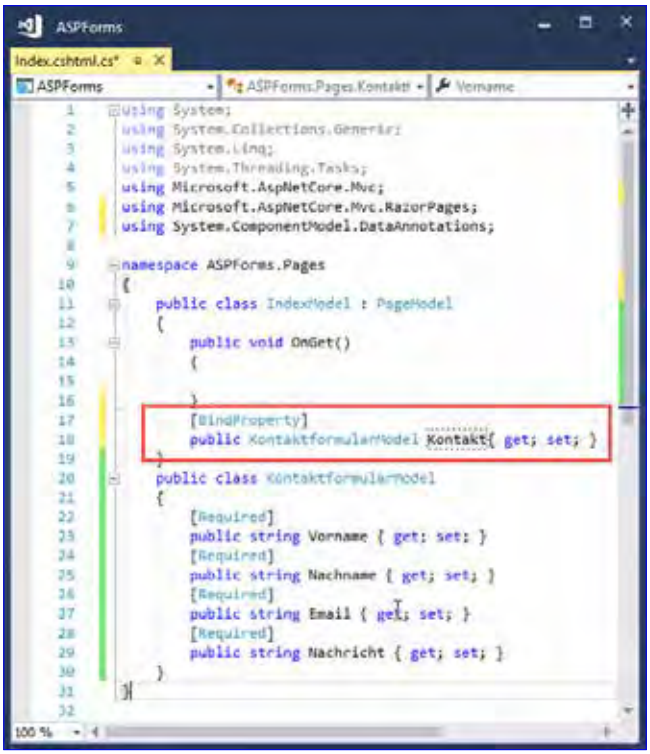


Bild 2: Aktueller Stand der Klassendatei **Index.cshtml.cs**

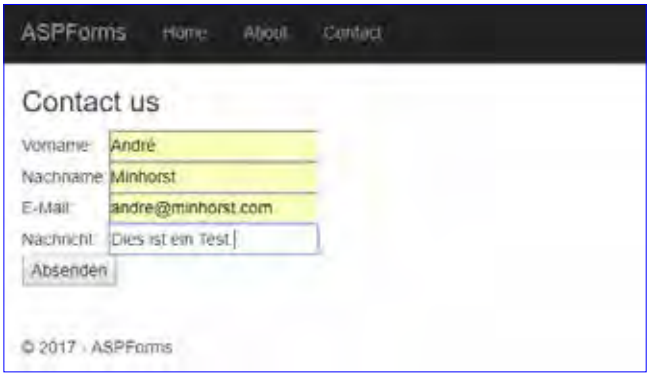


Bild 3: Start des ersten Beispiels

```
    </table>
</form>
```

Wenn wir nun auf Absenden klicken, geschieht allerdings noch nichts. Das liegt daran, dass wir nicht – wie früher üblich – die Zielseite angegeben haben, die nach dem Anklicken der **Submit**-Schaltfläche aufgerufen werden soll. Diese wurde sonst im Attribut **action** des **form**-Elements angegeben. Wir geben hingegen mit dem Attribut **method** nur die Art der Übermittlung der Inhalte des Formulars an, und zwar als **post** (es gibt auch noch **get**). Wenn Sie hier **get** angeben, werden die in das Formular eingegebenen Daten an die durch das Formular aufgerufene Seite über die URL übermittelt, für eine PHP-Seite also etwa mit **/index.php?vorname=andre&nachname=minhorst&email=andre@minhorst.com**. Der Nachteil dieser Version ist, dass die Anzahl der übermittelbaren Zeichen je nach Browser begrenzt ist. Deshalb verwenden wir **post** als **action**.

Auswertung des Formulars

Wie aber wollen wir das Formular nun nach dem Senden auswerten? Unter PHP beispielsweise gibt man eine Zielseite an, die dann den notwendigen Code enthält, um die entweder in der Variablen **$_GET** oder **$_POST** übermittelten Informationen zu verarbeiten. Unter ASP.NET Core läuft das ganz anders. Hier enthält die Code behind-Datei der Seite mit dem Formular eine Ereignismethode, welche die Daten auswertet. Sie brauchen also keine Zielseite für das Formular anzugeben, die Verarbeitung erfolgt in der Seite selbst beziehungsweise in ihrem serverseitigen Code. Die einfachste Version dieser Methode, die lediglich zu der Seite mit dem Namen **About.cshtml** weiterleitet, sieht wie folgt aus:

```
public async Task<IActionResult> OnPostAsync() {
    return RedirectToPage("About");
}
```

Wenn Sie die Anwendung nun starten, die Daten in das Formular eingeben und dann auf Absenden klicken, wird anschließend die Seite **About.cshtml** geladen.

Validieren

Und nun kommt ein wenig ASP.NET Core-Magie ins Spiel. Wir fügen nun eine **if**-Bedingung ein, welche die Eigenschaft **IsValid** des Objekts **ModelState** prüft und erneut die aktuelle Seite aufruft, wenn **IsValid** nicht den Wert **true** liefert:

```
public async Task<IActionResult> OnPostAsync() {
    if (!ModelState.IsValid) {
        return Page();
    }
    return RedirectToPage("About");
}
```

Wie funktioniert das? Nun: Wir haben ja mit dem **[Required]**-Element festgelegt, dass die Daten für die vier Felder der Klasse **KontaktformularModel** angegeben werden müssen. Diese werden bei der Prüfung durch **ModelState.IsValid** untersucht. Ist eines der Felder leer, hat **IsValid** den Wert **False** und man landet wieder auf der auslösenden Seite. Erst wenn alle Felder gefüllt sind, wird die Seite **About** mit der **RedirectToPage**-Methode aufgerufen.

Mail versenden

Um die Mail mit den Kontaktinformationen schließlich zu versenden, fügen wir der Methode **OnPostAsync** noch den Aufruf einer weiteren Methode namens **SendMail** hinzu:

```
public async Task<IActionResult> OnPostAsync() {
    if (!ModelState.IsValid) {
        return Page();
    }
    SendMail();
    return RedirectToPage("About");
}
```

Diese verwendet in diesem Fall nicht den Mail-Anbieter **Sendgrid** wie im Artikel **E-Mails per Sendgrid verschicken** beschrieben, sondern die Klasse **MailMessage** der Bibliothek **System.Net.Mail**, um die E-Mail zusammenzustellen und die Methode **Send** der Klasse **SmtpClient**, um die Mail zu versenden. Dabei nutzen wir auch schon die String-Interpolation, die wir im Artikel **Interpolierte Zeichenketten** beschrieben haben, um den Inhalt der E-Mail zusammenzustellen. Dabei pflegen wir die Inhalte des Objekts **Kontakt** auf Basis der Klasse **KontaktformularModel** direkt in geschweiften Klammern in den Mailtext ein:

```
private void SendMail() {
    var inhalt = $@"Hallo André,

hier kommt eine Kontaktanfrage:

Vorname: {Kontakt.Vorname}
Nachname: {Kontakt.Nachname}
E-Mail: {Kontakt.Email}
Nachricht: {Kontakt.Nachricht}";

    using (var mailMessage = new MailMessage(Kontakt.Email, "andre@minhorst.com")) {
        mailMessage.To.Add(new MailAddress("andre@minhorst.com"));
        mailMessage.From = new MailAddress(Kontakt.Email);
        mailMessage.Subject = "Kontaktanfrage";
        mailMessage.Body = inhalt;
        using (var smtpClient = new SmtpClient("minhorst.com")) {
            smtpClient.Send(mailMessage);
        }
    }
}
```

Im hinteren Teil der Methode erstellen wir ein neues Objekt namens **mailMessage** auf Basis der Klasse **MailMessage** und weisen dieser die Empfängeradresse, die Absenderadresse sowie den Betreff und für die Eigenschaft **Subject** den Inhalt der

Variablen **inhalt** zu. Dann verschickt sie die E-Mail über die Methode **Send** der Klasse **SmtpClient**, der wir beim Erstellen die Adresse des SMTP-Servers übergeben. In diesem Fall können wir den SMTP-Server problemlos nutzen, da wir eine Mail an eine der E-Mail-Adressen schicken, die von diesem Server verwaltet wird. Wollten Sie E-Mails an andere Adressen schicken, etwa um der Person, welche die Kontaktanfrage gesendet hat, eine Kopie davon zu schicken, sind Sie mit Sendgrid wohl besser bedient.

Optimierung: Validierung und Layout

Nun wollen wir noch etwas Komfort hinzufügen, beispielsweise indem wir die Validierung, die ja bereits durch die **[Required]**-Elemente vorgegeben ist, auch noch in die Benutzeroberfläche einbeziehen, sprich: ins Formular.

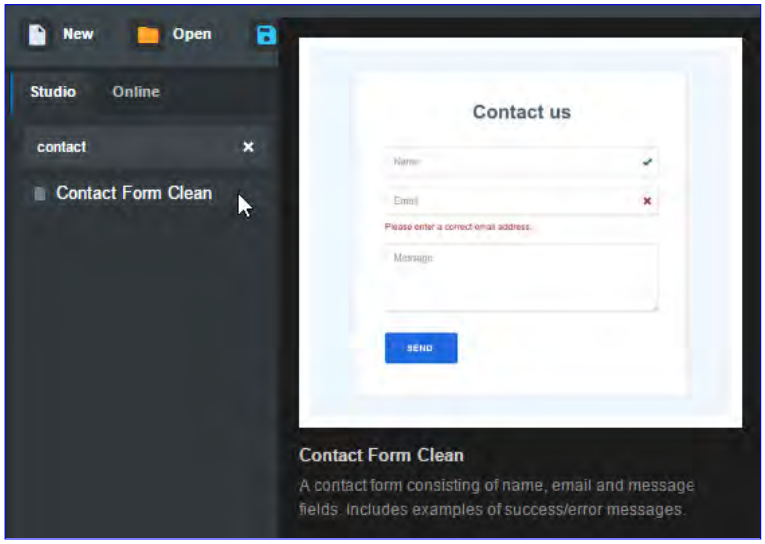


Bild 4: Vorschau eines Formulars

Gleichzeitig wollen wir auch noch die Optik anpassen und statt der Tabelle mit **div**-Elementen arbeiten, die ihre Format-Informationen aus entsprechenden CSS-Elementen beziehen.

Zum Erstellen des Designs wollen wir wieder Bootstrap Studio nutzen, das Sie bereits im Artikel **Webdesign mit Bootstrap Studio** kennengelernt haben.

Da wir ein Kontaktformular benötigen, können wir in Bootstrap Studio links oben im Feld **Search components** beispielsweise den Suchbegriff **contact** eingeben. Dies liefert nur einen Eintrag, der uns ein wenig schmal erscheint (siehe Bild 4).

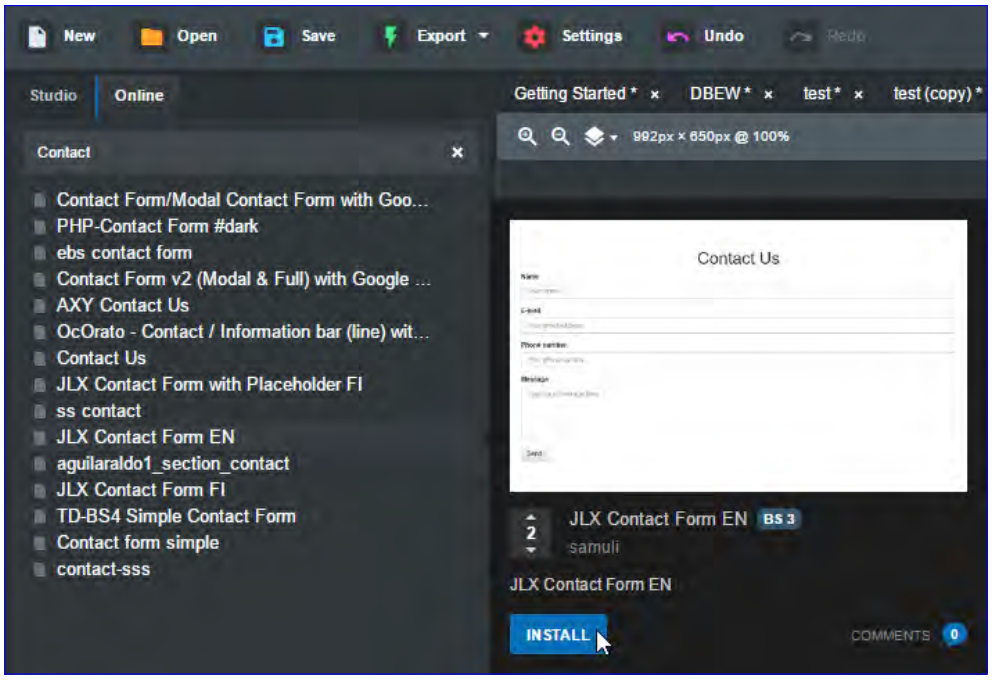


Bild 5: Alle Vorlagen mit dem Schlüsselwort **Contact**

Wir möchten lieber einen anderen Entwurf haben. Dazu gibt es den Bereich **Online**. Dieser zeigt standardmäßig die im Trend liegenden und die neuesten Vorlagen an. Wenn Sie hier den Suchbegriff **Contact** eingeben,

Razor Pages mit Datenbankanbindung

Nachdem Sie in weiteren Artikeln einige Grundlagen zur Programmierung von Razor Pages mit ASP.NET Core kennengelernt haben, geht es nun einen Schritt weiter: Wir wollen eine erste kleine Anwendung für den Zugriff auf die Daten einer Datenbank programmieren. Dazu beginnen wir mit einer kleinen Tabelle, für die wir eine Übersicht und eine Detailseite zum Ansehen und Bearbeiten der einzelnen Felder erstellen. Die Übersicht soll natürlich auch eine Möglichkeit zum Hinzufügen und Löschen der Datensätze bieten.

Voraussetzungen

Sie benötigen Visual Studio in der Version von 2017, die Sie mit den Komponenten wie im Artikel Visual Studio 2017 Community Edition beschrieben installieren.

Projekt erstellen

Wir erstellen ein Projekt auf Basis der Vorlage **Visual C#|Web|ASP.NET Core Webanwendung** (siehe Bild 1).

Bild 1: Projekt anlegen

Im folgenden Dialog wählen Sie den Eintrag **Webanwendung** aus, da wir hier schon einige Elemente bekommen, die wir sonst manuell nachreichen müssten (siehe Bild 2).

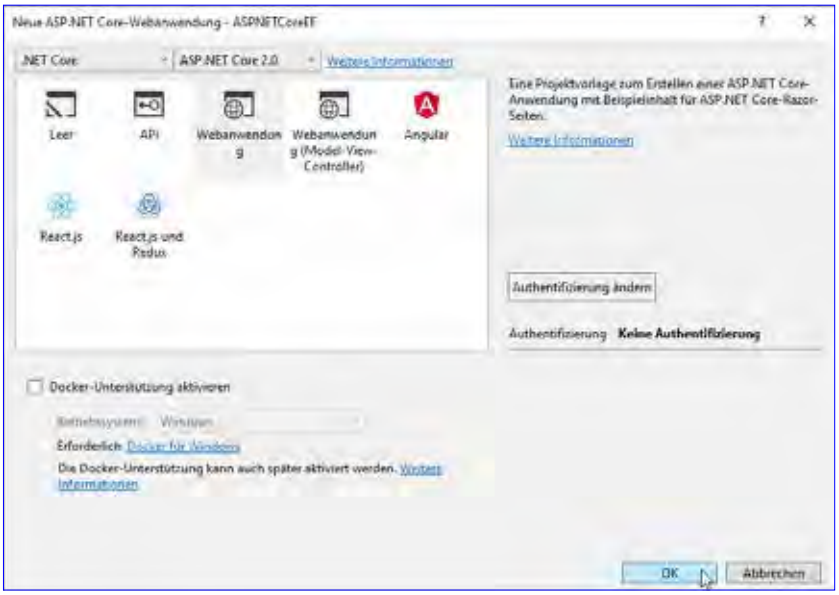


Bild 2: Projektvorlage definieren

Anpassen des Rahmens

Mit Rahmen meinen wir die Seite, welche die Navigationsleiste und den Fußbereich der Seite definiert. In unserem automatisch erstellten Demo-Projekt ist dies die Seite **_Layout.cshtml**. Wie die enthaltenen Elemente einer Webanwendung anpassen können, erfahren Sie beispielsweise im Artikel **ASP.NET Core-Anwendung anpassen**. In unserem Fall wollen wir die Bezeichnung unseres Projekts, das an verschiedenen Stellen in der Datei **_Layout.cshtml** gelandet ist, durch die Bezeichnung unserer Anwendung ersetzen – also beispielsweise **Kundenverwaltung**. Außerdem wollen wir das Navigationsmenü so ändern, dass es hinter den Einträgen **Home** und **About** den Eintrag **Kunden** anzeigt.

Damit dies geschieht, ändern wir in Bild 3 markierten Zeilen in der Datei **_Layout.cshtml**. Die eingerahmten Stellen markieren die zu ändernden Stellen, der Pfeil zeigt den Befehl **@RenderBody()**, an dem später die eigentlichen Inhalte eingefügt werden.

Den Inhalt der Index-Seite, also der Seite, die direkt beim Starten der Anwendung im Browser angezeigt wird, können Sie auch anpassen. Den ersten **div**-Teil mit **id="myCarousel"** entfernen wir komplett. Im folgenden **div**-Element entfernen wir die hinteren Auflistungen. Den vorderen Teil können wir wie folgt anpassen. Dabei wollen wir eine kleine Übersicht über die enthaltenen Funktionen bieten. Der Code sieht anschließend wie folgt aus:

Bild 3: Datei **_Layout.cshtml** anpassen

```
@page
@model IndexModel
@{
    ViewData["Title"] = "Home page";
}
<div class="row">
    <div class="col-md-3">
        <h2>Die Beispielanwendung Kundenverwaltung</h2>
        <p>Diese Beispielanwendung zeigt, wie Sie eine Demo-Anwendung um Datenbankfunktionen erweitern. Dazu gehören:</p>
        <ul>
            <li>Übersichten für Kunden und andere Entitäten mit Schaltflächen zum Anzeigen oder Löschen von Kunden</li>
            <li>Anzeige der Kundendetails zum Bearbeiten der Daten</li>
```

```
            <li>Anzeige der Kundendetails zum Anlegen neuer Kundendatensätze</li>
        </ul>
    </div>
</div>
```

Wenn Sie die Anwendung danach starten, zeigt der Browser die Seite aus Bild 4 an. Für die wenigen Änderungen, die wir vorgenommen haben, sieht diese schon recht ordentlich aus. Ein Klick auf den Link Kunden in der Navigationsleiste zeigt nun leider noch keine Kundenseite an, was wir als Nächstes in Angriff nehmen werden.

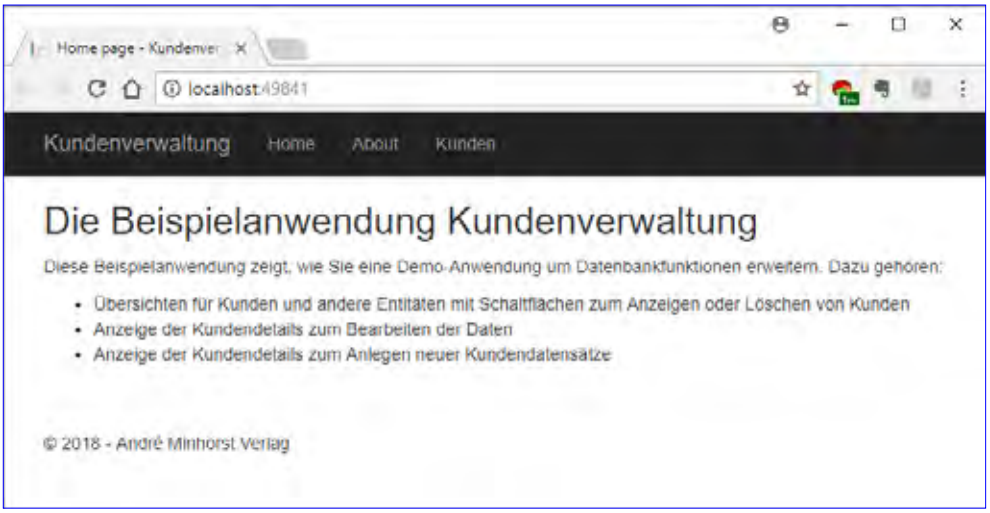


Bild 4: Die angepasste Startseite

Kunden-Klasse hinzufügen

Nun wollen wir uns um die anzuzeigenden Daten kümmern. Dazu benötigen wir eine Datenbank und ein entsprechendes Entity Data Model mit den Klassen für die einzelnen Entitäten. Wir verwenden den Code First-Ansatz, bei dem wir erst die Entitätsklassen definieren und daraus dann automatisch die Datenbank erstellen lassen. Für die Entitätsklassen legen wir direkt unterhalb des Projektordners ein neues Verzeichnis namens **Models** an.

In diesem Verzeichnis erstellen wir mit dem Kontextmenü-Eintrag **Hinzufügen|Klasse...** eine neue Klassendatei, der wir im folgenden Dialog **Neues Element hinzufügen** den Namen **Customer** geben. Wir könnten auch die deutsche Bezeichnung verwenden, aber beim Entity Framework ist es einfacher, wenn man die Bezeichnungen in englischer Sprache angibt. So kann man leichter Singular durch einfaches Anhängen des Buchstaben **s** in Plural verwandeln. Außerdem muss man dort nicht aufpassen, ob eine Bezeichnung gleichzeitig Singular und Plural ist (im Deutschen beispielsweise **Artikel**!).

Die Klasse sieht wie folgt aus und wird in der Klassendatei **Customer.cs** angelegt:

```
namespace ASPNETCoreEF.Models {
    public class Customer {
        public int ID { get; set; }
        public string FirstName { get; set; }
        public string LastName { get; set; }
        public string Street { get; set; }
        public string Zip { get; set; }
```

```
        public string City { get; set; }
    }
}
```

Das Feld **ID** soll der Primärschüssel der noch zu erstellenden Tabelle sein, in der die Daten zu den Kunden gespeichert werden. Die übrigen Felder benötigen keine weitere Erläuterung. Fremdschlüsselfelder und weitere Tabellen lassen wir aus Gründen der Übersicht zunächst außen vor.

Beachten Sie, dass durch das Anlegen der Klassendatei im Verzeichnis **Models** auch der Namespace auf **ASPNETCoreEF.Models** eingestellt wurde (**ASPNETCoreEF** ist der Name des Projekts und somit auch als Namespace des Projekts voreingestellt).

Datenbankkontext erstellen

Nun benötigen wir noch die Klasse, welche die Informationen der einzelnen Entitätsklassen – hier nur eine – zusammenfasst und Informationen zum Erstellen des Datenmodells speichert. Dazu fügen wir dem Projekt zunächst einen weiteren neuen Ordner namens **Data** hinzu. Hier legen wir eine neue Klasse namens **CustomerManagementContext** an. Dieser fügen wir im Kopf der Klassendatei zunächst einen Verweis auf den Namespace **Microsoft.EntityFrameworkCore** hinzu:

```
using Microsoft.EntityFrameworkCore;
```

Da die **Customer**-Klasse sich im Verzeichnis **Models** und somit auch im Namespace **ASPNETCoreEF.Models** befindet, fügen wir auch diesen Namespace noch hinzu:

```
using ASPNETCoreEF.Models;
```

Die Klasse soll die Schnittstelle **DbContext** implementieren. Außerdem erhält Sie ein **DbSet**-Objekt namens **Customers**, welches Elemente des Typs **Customer** aufnehmen soll:

```
public class CustomerManagementContext : DbContext {
    public CustomerManagementContext(DbContextOptions<CustomerManagementContext> options) : base(options) {

    }
    public DbSet<Customer> Customers { get; set; }
}
```

Wir überschreiben noch die Methode **OnModelCreating** und geben dort an, dass die Tabelle für die Kunden nicht **Customer** heißen soll, sondern im Plural **Customers** angelegt werden soll:

```
protected override void OnModelCreating(ModelBuilder modelBuilder) {
    modelBuilder.Entity<Customer>().ToTable("Customers");
}
```

DbContext beim Anwendungsstart registrieren

Damit die Datenbank über das **DbContext**-Objekt gleich nach dem Start zur Verfügung steht, fügen wir der Methode **ConfigureServices** der Datei noch eine entsprechende Anweisung hinzu. Die Datei **Startup.cs** erhält jedoch zuvor zwei zusätzliche Verweise auf die benötigten Namespaces:

```
using Microsoft.EntityFrameworkCore;
using ASPNETCoreEF.Data;
```

Dann ergänzen wir **ConfigureServices** wie folgt:

```
public void ConfigureServices(IServiceCollection services) {
    services.AddMvc();
    services.AddDbContext<CustomerManagementContext>(options =>
        options.UseSqlServer(Configuration.GetConnectionString("DefaultConnection")));
}
```

Dadurch greift das Projekt auf die Verbindungszeichenfolge zu, die wir noch in der Datei **appsettings.json** speichern müssen. Diese erweitern wir wie folgt:

```
{
  "ConnectionStrings": {
    "DefaultConnection": "Server=(localdb)\\mssqllocaldb;Database=CustomerManagement;ConnectRetryCount=0;
        Trusted_Connection=True;MultipleActiveResultSets=true"
  },
  "Logging": {
    "IncludeScopes": false,
    "LogLevel": {
      "Default": "Warning"
    }
  }
}
```

Hier legen wir in der Einstellung **ConnectionStrings** fest, dass die Standardverbindungszeichenfolge den Server **localdb** verwenden soll und der Datenbankname **CustomerManagement** lautet. Außerdem soll Windows-Authentifizierung für den Zugriff auf die Datenbank verwendet werden (**Trusted_Connection=True**).

Ein paar Daten vorbereiten

Nun wollen wir noch dafür sorgen, dass die Tabelle **Customers** beim Start der Anwendung nicht völlig leer ist. Daher fügen wir der Anwendung im Verzeichnis **Data** eine neue Klassendatei namens **DbInitializer.cs** hinzu. Diese füllen wir wie folgt:

```
public class DbInitializer {
```

```
    public static void Initialize(CustomerManagementContext dbContext) {
        dbContext.Database.EnsureCreated();
        if (dbContext.Customers.Any()) {
            return;
        }
        var customers = new Customer[] {
            new Customer{FirstName="André", LastName="Minhorst", Street="Borkhofer Str. 17", Zip="47137", City="Duisburg"},
            new Customer{ FirstName="Klaus", LastName="Müller", Street="Teststr. 1", Zip="12345", City="Berlin" },
            new Customer{ FirstName="Herbert", LastName="Meier", Street="Beispielweg 2", Zip="23456", City="Bremen"}
        };
        foreach (Customer c in customers) {
            dbContext.Customers.Add(c);
        }
        dbContext.SaveChanges();
    }
}
```

Die Klasse enthält eine Methode namens **Initialize**, welche mit dem Parameter **dbContext** ein Objekt des Typs **CustomerManagementContext** entgegennimmt. Sie prüft zunächst ob die Datenbank bereits erstellt wurde und erledigt dies gegebenenfalls im gleichen Schritt mit der Methode **EnsureCreated**. Dann prüft sie, ob die **Customers**-Tabelle bereits Daten enthält. Falls ja, bricht sie an dieser Stelle ab. Anderenfalls erstellt sie ein Array mit drei Elementen des Typs **Customer**. Die folgende **foreach**-Schleife durchläuft dieses Array und fügt die einzelnen Elemente des Typs **Customer** zum **DbSet** namens **Customers** des **dbContext** hinzu. Die Änderungen werden schließlich mit der **SaveChanges**-Methode gespeichert.

Nun müssen wir noch dafür sorgen, dass diese Methode zum Erstellen beziehungsweise Füllen der Datenbank auch ausgeführt wird. Dazu ändern wir in der Klassendatei **Program.cs** die Methode **Main** wie folgt:

```
public static void Main(string[] args) {
    //            BuildWebHost(args).Run();
    var host = BuildWebHost(args);
    using (var scope = host.Services.CreateScope()) {
        var services = scope.ServiceProvider;
        var dbContext = services.GetRequiredService<CustomerManagementContext>();
        DbInitializer.Initialize(dbContext);
    }
    host.Run();
}
```

Dabei kommentieren wir die dort bereits vorhandene Anweisung aus. Diese ein neues Webhost-Objekt und führte direkt seine **Run**-Methode aus. Hier wollen wir das Erstellen der Datenbank zwischenschalten. Dazu teilen wir die Anweisung auf, sodass diese zunächst das neue Objekt mit **BuildWebHost** erstellt und in der neuen Variable namens **host** speichert. Die **Run**-Methode

für die Variable **host** wird dann als letzte Anweisung ausgeführt. Dazwischen fügen wir ein paar neue Anweisungen ein. Die erste erstellt ein neues **Scope**-Element und speichert es in der Variablen **scope**. Über die Eigenschaft **ServiceProvider** holt die Prozedur dann ein **Service**-Objekt und speichert es in der Variablen **services**. Diees stellt dann über die Methode **GetRequiredService** mit dem Typ **CustomerManagementContext** den Datenbankkontext zur Verfügung und speichert diesen in der Variablen **dbContext**. Dieses Objekt übergeben wir dann der Methode **Initialize** der zuvor programmierten Klasse **DbInitializer**.

Damit alle als fehlerhaft markierten Stellen verschwinden, müssen Sie wieder die folgenden Namespaces referenzieren:

```
using Microsoft.Extensions.DependencyInjection;
using ASPNETCoreEF.Data;
```

Datenbank erstellen

Damit ist es soweit: Sie können die Anwendung nun starten und damit die Datenbank erstellen. Sobald der Browser aufgerufen wurde und die Startseite der Anwendung anzeigt, können Sie diese wieder beenden. Wir blenden dann den **SQL Server-Objekt-Explorer** ein (Menüeintrag **Ansicht|SQL Server-Objekt-Explorer**) und klappen die Einträge wie in Bild 5 auf. Wenn die Datenbank **CustomerManagement** dort angezeigt wird, navigieren Sie weiter zur Tabelle **dbo.Customers** und wählen den Kontextmenü-Eintrag **Daten anzeigen** aus. Damit sollten auch die soeben per Code hinzugefügten Datensätze erscheinen.

Die beiden Datenbankdateien **CustomerManagement.mdf** und **CustomerManagement.ldf** wurden auf dem lokalen Rechner im Verzeichnis **C:\Users\<Benutzername>** gespeichert. Dabei wurde automatisch der Name als Datenbankname verwendet, den wir in der Verbindungszeichenfolge angegeben haben. Der Tabellenname entspricht wie von uns gewünscht der Plural-Form der verwendeten Klasse **Customer**, also **Customers**. Die Eigenschaftsnamen der Klasse **Customer** wurden als Feldnamen verwendet und das Feld mit dem Namen **ID** wurde automatisch als Primärschlüsselfeld festgelegt.

Kundenübersicht hinzufügen

Da wir neben den Kunden später noch weitere Daten zur Anwendung hinzufügen werden, legen wir für jede Entität einen eigenen Ordner an. Als Erstes benötigen wir also einen Ordner für Kunden, der die Übersicht, die Detailseite et cetera aufnehmen soll. Diesen Ordner legen wir unterhalb des Ordners **Pages** an, und zwar über dessen Kontextmenü-Eintrag **Hinzufügen|Neuer Ordner**. Der neue Ordner soll analog zur englischen Bezeichnung der Tabelle **Customers** heißen.

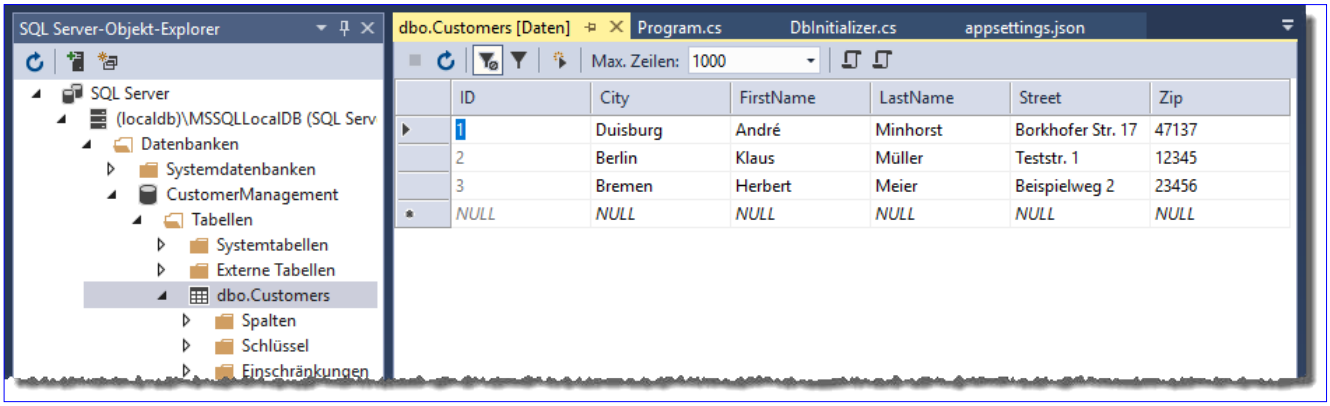


Bild 5: Die neue Datenbank mit den frischen angelegten Datensätzen

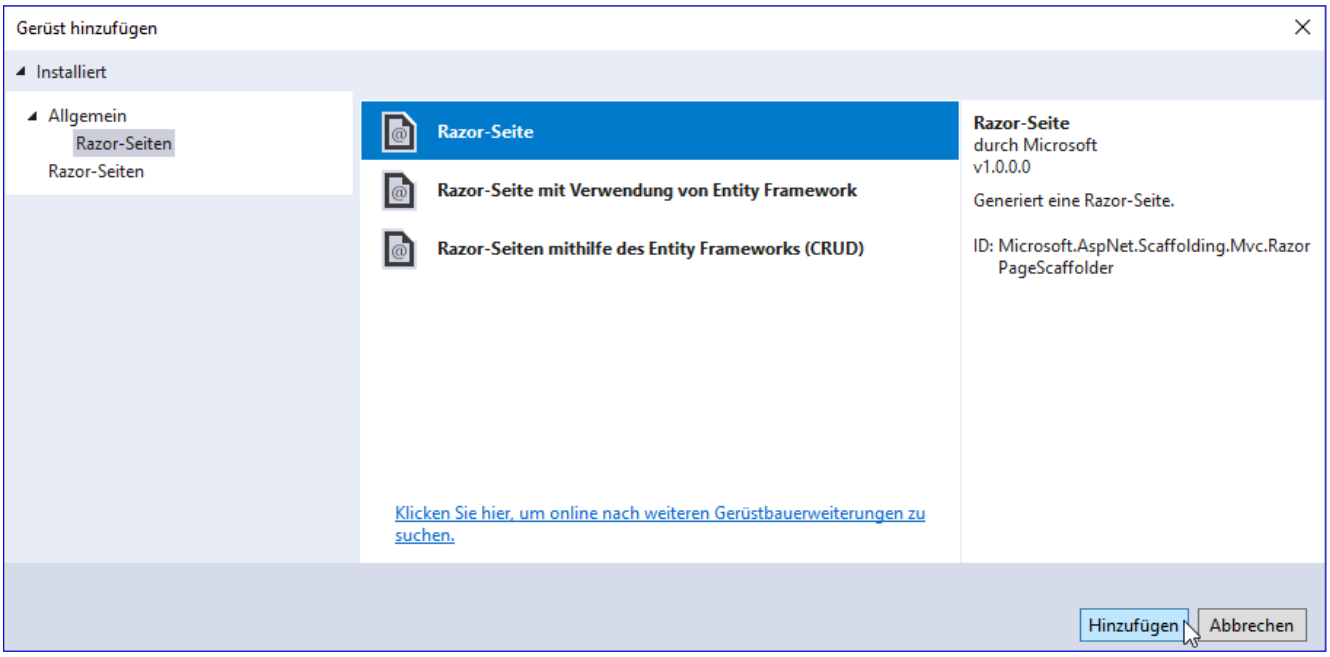


Bild 6: Anlegen einer neuen Razor-Seite

Diesem Ordner fügen wir gleich eine neue Razor-Seite hinzu, die wir **Index.cshtml** nennen. Eine neue Razor-Seite fügen Sie dem Ordner hinzu, indem Sie aus seinem Kontextmenü den Eintrag **Hinzufügen|Razor-Seite...** auswählen. Im folgenden Dialog finden Sie dann drei Möglichkeiten (siehe Bild 6):

- **Razor-Seite**
- **Razor-Seite mit Verwendung von Entity Framework**
- **Razor-Seiten mithilfe des Entity Frameworks (CRUD)**

Einfache Razor-Seite anlegen

Wählen Sie die erste Option wählen, geben Sie im folgenden Schritt einfach den Namen der gewünschten Seite an und erstellen diese. Damit erhalten Sie dann eine fast leere Razor-Seite mit zugehöriger **.cshtml.cs**-Klassendatei, die Sie weitgehend selbst füllen müssen. Diese Option ist interessant, wenn Sie komplett neue Seiten aufbauen möchten. Die anderen beiden Optionen hören sich aber wesentlich interessanter an!

Razor-Seite mit Verwendung von Entity Framework

Die zweite Option unterscheidet sich vom Titel her von der dritten Option, weil sie offenbar nur eine Seite anlegt und nicht mehrere. Schauen wir uns diese an. Nach der Auswahl erscheint der Dialog aus Bild 7. Hier geben wir zunächst den Namen der Razor-Seite an. Diese soll Index heißen, weil wir zunächst eine Übersicht der Kunden erstellen wollen. Warum nennen wir die Seite dann nicht **CustomerOverview** oder ähnlich? Weil die Übersicht die Standardseite sein soll, die auch beim Aufrufen des Ordners **/Customer** bereits erscheint. Wird ein Ordner in der URL angegeben und keine spezielle Datei, ruft die Anwendung die Datei **Index.cshtml** auf, sofern diese in dem Ordner enthalten ist.

Direkt unter dem Namen wählen Sie aus, welche Vorlage für das Erstellen der Seite genutzt werden soll. Es gibt fünf Möglichkeiten zur Auswahl:

- **Create**: Anzeige eines leeren, neuen Datensatzes
- **Delete**: Bestätigungsseite zum Löschen eines Datensatzes, nachdem auf der Übersichtsseite die Löschen-Schaltfläche angeklickt wurde
- **Details**: Detailansicht eines einzelnen Datensatzes ohne die Möglichkeit zum Bearbeiten
- **Edit**: Detailansicht eines einzelnen Datensatzes zum Bearbeiten
- **List**: Auflistung der enthaltenen Datensätze

Wir wählen hier zunächst den Eintrag **List** aus, da wir ja eine Übersicht erstellen wollen.

Das nächste Auswahlfeld dient der Auswahl der Klasse, für welche die Seite erstellt werden soll. In unserem Fall ist die Klasse **Customer** die einzige bisher erstellte Klasse, also wählen wir diese aus (siehe Bild 8).

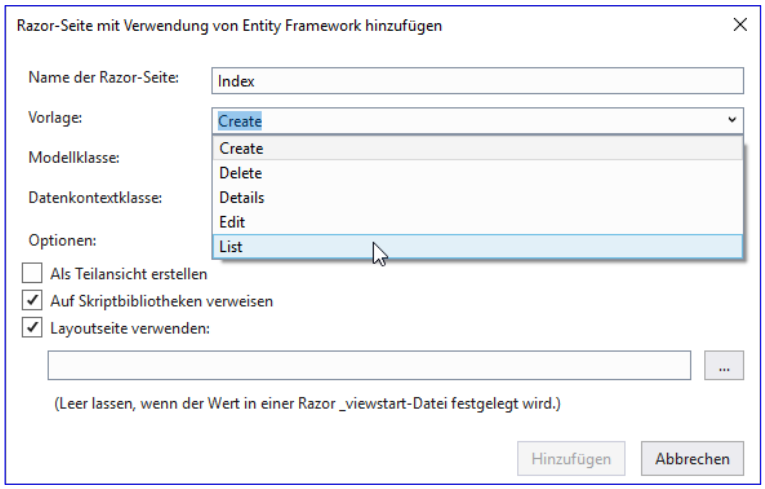


Bild 7: Auswahl des Inhalts der Seite

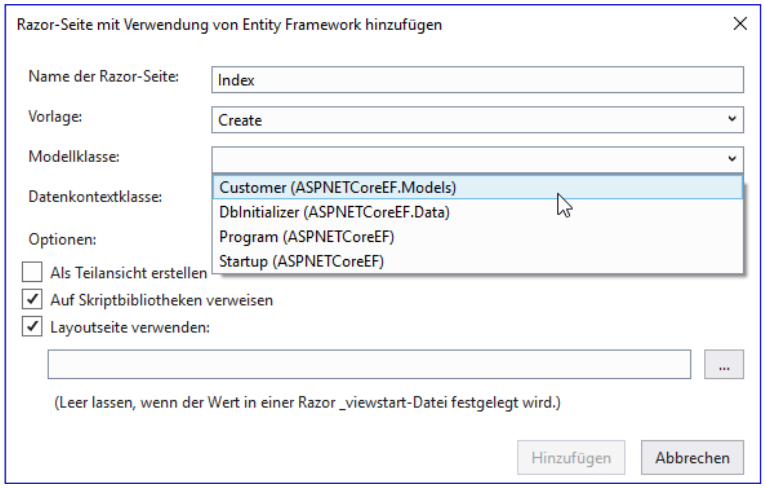


Bild 8: Auswählen des Inhalts

Nach dem Bestätigen der Auswahl durch einen Klick auf die Schaltfläche **Hinzufügen** arbeitet Visual Studio ein paar Sekunden und erstellt dabei die neue Razor-Seite. Schauen wir uns zunächst den Code an, den Visual Studio für die **.cshtml**-Datei hinterlegt hat (wir teilen diesen in den folgenden Abschnitten auf, um ihn besser erläutern zu können). Der Code beginnt mit dem obligatorischen **@page**-Element, das die Seite als MVC-Aktion markiert. Die folgende **@model**-Zeile legt fest, welches Model für die Seite verwendet wird, also welche Klasse die Eigenschaften, Methoden und Ereignisse für die Seite enthält. In diesem Fall handelt es sich um die Klasse **IndexModel**, die in der Datei **Index.cshtml.cs** enthalten ist. Der Namespace sowie die Verzeichnisse werden in der **@model**-Anweisung ebenfalls vermerkt, sodass ein Eintrag mit dem Wert **ASPNETCoreEF.Pages.Customers.IndexModel** herauskommt. Schließlich folgt noch ein Razor-Abschnitt, der den Titel auf den Wert **Index** festlegt:

```
@page
@model ASPNETCoreEF.Pages.Customers.IndexModel
@{ ViewData["Title"] = "Index"; }
```

Damit kommen wir zum HTML-Teil, der allerdings von Razor-Elementen durchsetzt ist. Bevor wir uns diesen ansehen, werfen wir noch einen Blick auf die Seite, wie Sie nach dem Starten der Anwendung im Webbrowser angezeigt wird (siehe Bild 9).

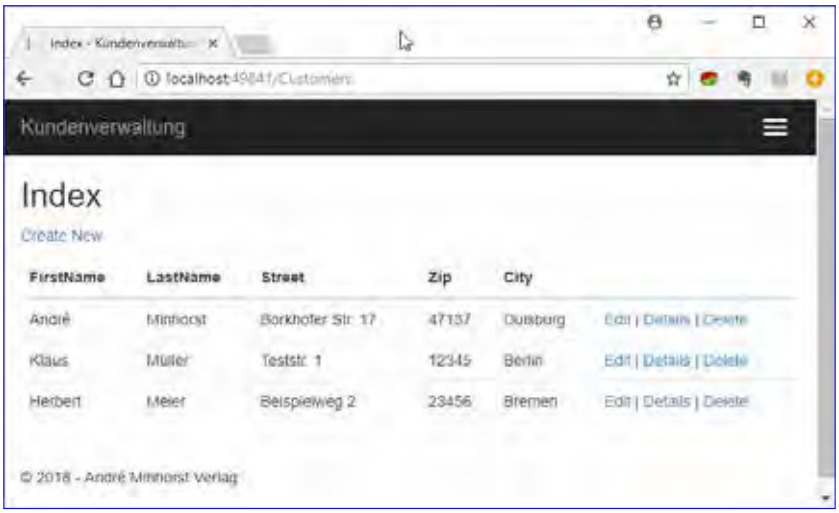


Bild 9: Die Kunden-Übersicht

Sie finden hier oben einen Link zum Anlegen eines neuen Datensatzes. Die Auflistung der vorhandenen Datensätze bietet rechts jeweils Links zum Bearbeiten, Betrachten und Löschen der Einträge. Hier sehen wir gleich, dass alle vom System erzeugten Elemente in englischer Sprache erscheinen – das müssen wir später noch ändern.

Damit Sie die Seite über den Menüeintrag **Kunden** anzeigen können, müssen Sie den entsprechenden Link in der Datei **_Layout.cshtml** noch wie folgt anpassen – im ersten Entwurf lautete auch das Ziel des Links noch auf **/Kunden**, aber wir verwenden ja nun die englische Version:

```
<li><a asp-page="/Customers/Index">Kunden</a></li>
```

Der Code enthält zunächst die **h2**-Überschrift und dann bereits den Link zum Anlegen eines neuen Eintrags. Dieser verweist auf die Seite namens **Create** (oder die Seite **Index.cshtml** in einem Verzeichnis namens **Create**). Diese ist natürlich noch nicht vorhanden, da wir nur eine einzelne Seite zur Anzeige der Übersicht erstellt haben.

Darunter folgt ein **table**-Element, dem zunächst die Überschriften untergeordnet sind. Die Überschriften werden dabei aus einem Ausdruck wie **@Html.DisplayNameFor(model => model.Customer[0].FirstName)** gewonnen. Was bedeutet das? **Html.DisplayNameFor** ist ein sogenannter HTML Helper (mehr dazu im Artikel **ASP.NET Core: HTML Helper**). Damit greifen Sie auf die Elemente des für die Seite angegebenen Modells zu, also hier der Klasse **IndexModel**, die wir weiter unten vorstellen. Hier nutzen wir die Methode **DisplayNameFor** für die Felder des ersten Elements des Objekts **Customer** dieser Klasse, um deren Anzeigenamen zu ermitteln. Diese landen dann in den Spaltenköpfen der HTML-Tabelle:

```
<h2>Index</h2>
<p><a asp-page="Create">Create New</a></p>
<table class="table">
    <thead>
        <tr>
            <th>@Html.DisplayNameFor(model => model.Customer[0].FirstName)</th>
            <th>@Html.DisplayNameFor(model => model.Customer[0].LastName)</th>
            <th>@Html.DisplayNameFor(model => model.Customer[0].Street)</th>
```

Authentifizierung unter ASP.NET Core

In weiteren Artikeln dieser Ausgabe haben wir uns mit ASP.NET Core und den Razor Pages beschäftigt. Der vorliegende Artikel zeigt, wie Sie beim Erstellen eines neuen Projekts automatisch die Elemente für eine Benutzerverwaltung zum Projekt hinzufügen können. Damit werden automatisch etwa die notwendigen Datenbanktabellen angelegt und Elemente für die Anmeldung und die Registrierung hinzugefügt.

Wenn Sie eine Webanwendung bauen wollen, die dynamische Inhalte anzeigt, sind Sie mit ASP.NET Core und den Razor Pages aktuell auf der richtigen Seite. In vielen Fällen sollen solche Webanwendungen nicht nur dynamische Inhalte anzeigen, sondern diese unter Umständen teilweise nur angemeldeten Benutzern bereitstellen. Das ist ja beispielsweise auch bei den Artikeln und Downloads zu diesem Magazin der Fall – als Kunde können Sie sich im Shop unter **shop.minhorst.com** anmelden, nachdem Sie sich registriert haben, und erhalten dann Zugriff auf einen Bereich namens **Meine Sofortdownloads**. Oder Sie schauen die einzelnen Artikel im HTML-Format im Know-how-Bereich der Seite ein. Auch hier wird geprüft, ob überhaupt ein Benutzer angemeldet ist und falls ja, ob er das Produkt mit den gewünschten Inhalten abonniert hat.

Wenn Sie selbst eine Internetseite mit einem geschützten Bereich programmieren wollen, auf den nur angemeldete Benutzer zugreifen können, haben Sie eine Menge Arbeit vor sich. Sie müssen Datenbanktabellen erstellen, welche die Benutzer und ihre Berechtigungen verwalten und gegebenenfalls auch noch Benutzergruppen definieren. Sie benötigen eine Seite, auf der Benutzer sich registrieren können und eine weitere Seite, auf der sie sich später erneut einloggen können. Die Registrierungsseite muss Automatismen aufweisen, die prüfen, ob der Benutzername bereits vorhanden ist und ob das Kennwort den gewünschten Konventionen entspricht – zum Beispiel hinsichtlich der Mindestanzahl von Zeichen und Sonderwünschen wir das Vorkommen von Zahlen, großen Buchstaben oder Sonderzeichen. Dann muss die Seite dafür sorgen, dass der neue Benutzer angelegt wird.

Die Login-Seite soll die Daten des registrierten Benutzers, also Benutzername und Kennwort, entgegennehmen und für die aktuelle Session speichern, welcher Benutzer angemeldet ist. Außerdem soll die Anwendung jederzeit prüfen können, ob aktuell ein Benutzer angemeldet ist.

Wenn Sie das alles selbst programmieren wollen, haben Sie eine Weile zu tun. Doch warum sollte man, wenn es dazu fertige Pakete

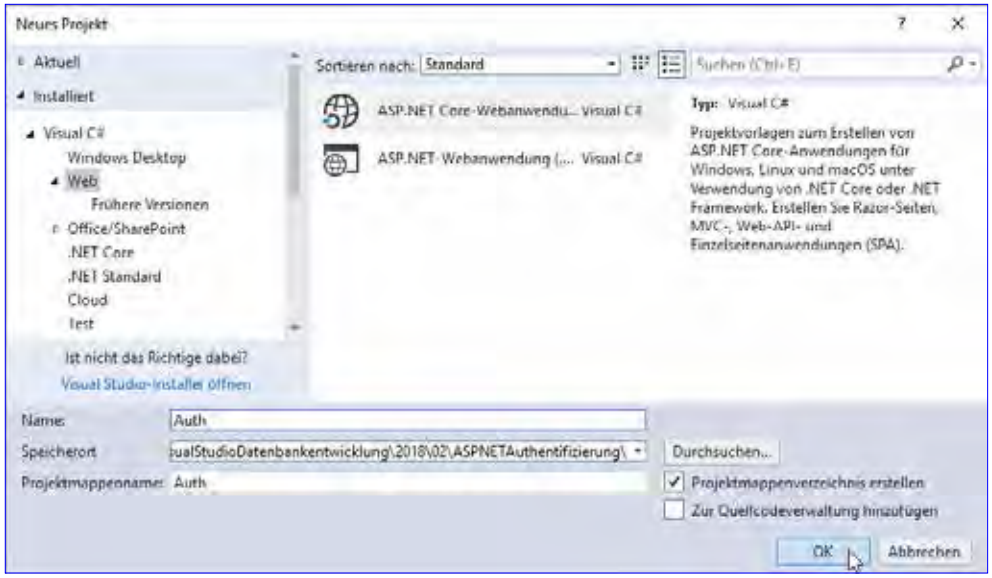


Bild 1: Anlegen eines neuen Projekts auf Basis der Vorlage **ASP.NET Core Webanwendung**

gibt, die Sie direkt beim Anlegen des Projekts zu diesem hinzufügen können?

Projekt mit Authentifizierung anlegen

Um ein Projekt direkt mit Authentifizierung anzulegen, erstellen Sie dieses zunächst wie gewohnt über den Dialog **Neues Projekt** unter dem gewünschten Namen und Verzeichnis (siehe Bild 1).

Im nächsten Schritt wählen Sie als Projekttyp den Eintrag **Webanwendung** aus. Hier finden Sie außerdem die Schaltfläche **Authentifizierung ändern** (siehe Bild 2). Bisher zeigt der Text darunter noch den Wert **Keine Authentifizierung** als Authentifizierungsmethode an.

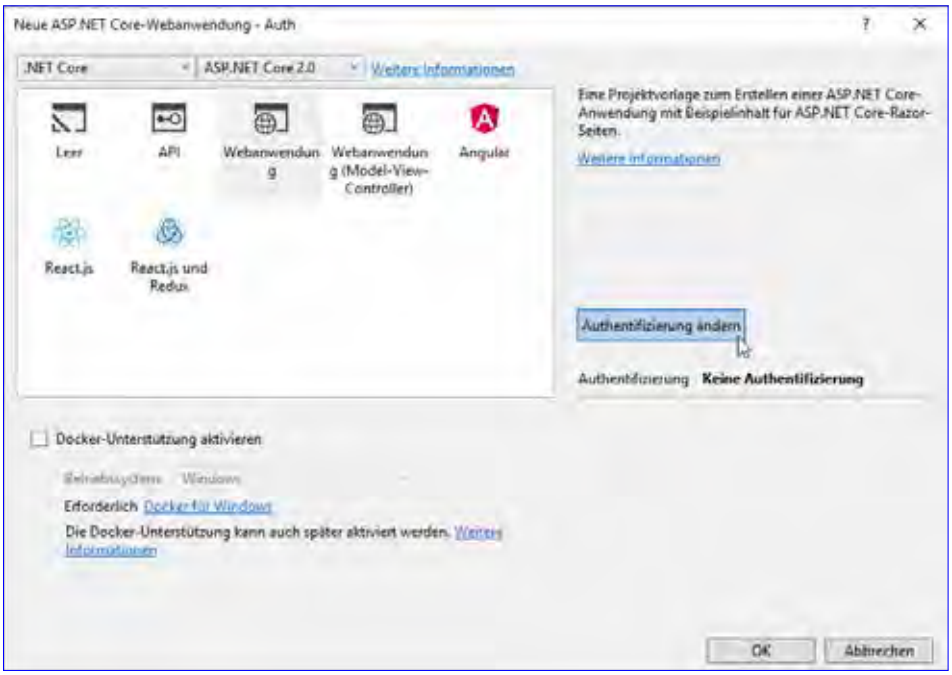


Bild 2: Aktivieren des Eintrags **Webanwendung** und Klick auf **Authentifizierung ändern**

Das ändern wir nach dem Anklicken der Schaltfläche im Dialog **Authentifizierung ändern** (siehe Bild 3). Hier haben Sie die folgenden Wahlmöglichkeiten:

Bild 3: Einstellen der Authentifizierungsmethode

- **Keine Authentifizierung**: Es werden keine Elemente zur Unterstützung einer Authentifizierung angelegt.
- **Einzelne Benutzerkonten**: Die Benutzerkonten werden lokal gespeichert, in diesem Fall in der Datenbank der Anwendung.
- **Geschäfts-, Schul oder Unikonten**: Die Authentifizierung erfolgt über Active Directory, Microsoft Azure Active Directory oder Office 365.
- **Windows-Authentifizierung**: Die Authentifizierung basiert auf den Windows-Konten. Diese Methode ist etwa für Intranet-Anwendungen geeignet.

Wir wollen eine Webanwendung für beliebig viele Benutzer programmieren und nutzen daher die Option **Einzelne Benutzerkonten**. Dabei behalten wir die Einstellung **In-App-Speicherung von Benutzerkonten** bei, da wir eine lokale Datenbank für die Benutzerdaten nutzen wollen.

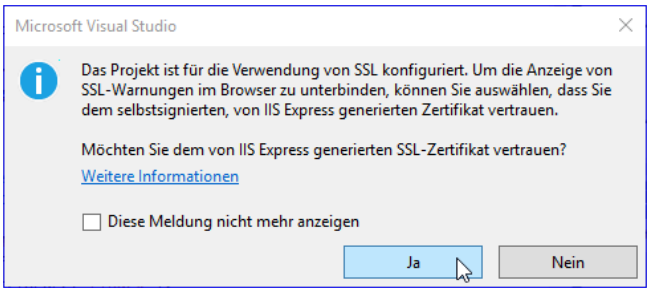


Bild 4: Hinweis auf die Verwendung von SSL

Was ist nun der Unterschied zu einer Anwendung ohne Authentifizierung? Dazu starten wir die Anwendung zunächst einmal. Dabei erhalten wir die Meldung aus Bild 4. Im Gegensatz zur Webanwendung ohne Authentifizierung verwendet diese hier eine SSL-Verschlüsselung. Das macht Sinn, denn niemand möchte Kennwörter im Klartext über das Internet verschicken.

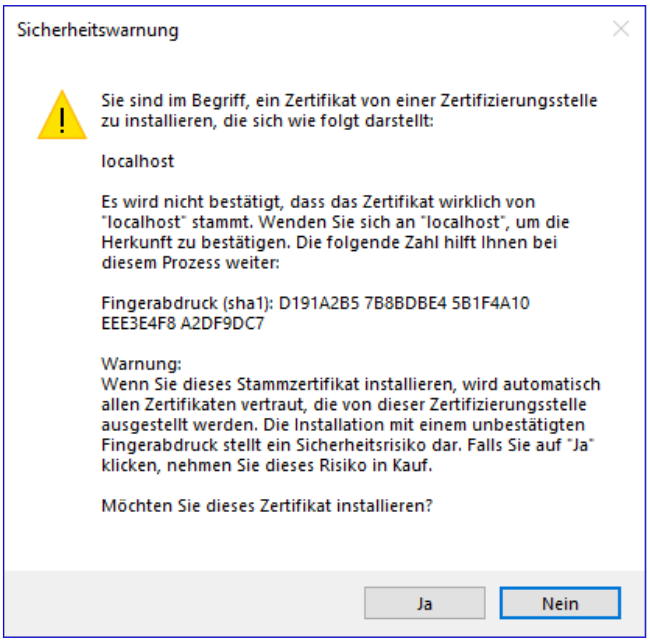


Bild 5: Installation eines Zertifikats auf dem Localhost

Die nächste Meldung kündigt an, dass für den Rechner **localhost**, also den aktuellen Rechner, ein Zertifikat installiert wird (siehe Bild 5).

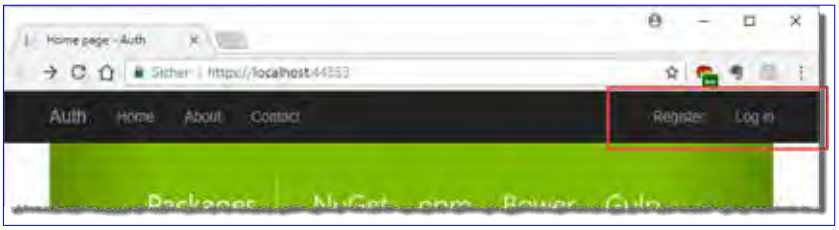


Bild 6: Zwei neue Einträge zum Registrieren und Einloggen

Nach dem Starten der Webanwendung zeigen sich zunächst keine offensichtlichen Änderungen. Erst ein Blick nach oben rechts liefert zwei Menüeintrage, die bei dem Projekt ohne Authentifizierung noch nicht vorhanden waren – nämlich **Register** und **Log in** (siehe Bild 6).

Ein Klick auf den Link **Register** zeigt dann das Formular aus Bild 7 an. Hier finden Sie die üblichen Textfelder zur Eingabe der E-Mail-Adresse sowie Textfelder zur zweifachen Eingabe des Kennworts. Hier finden wir bereits einige Mechanismen zur Prüfung der Validität des Kennworts.

Zum Beispiel meckert das Formular, wenn Sie ein Kennwort eingeben, das weniger als sechs Zeichen enthält oder wenn die beiden Kennwörter nicht übereinstimmen.

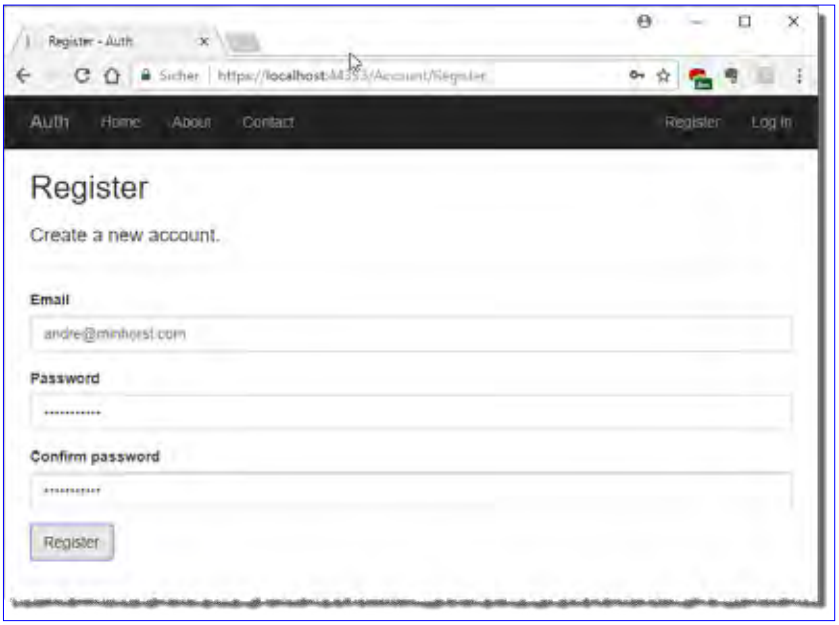


Bild 7: Registrieren eines neuen Benutzers

Weitere Validierungsmeldungen treten erst nach dem Bestätigen der Eingabe über die Schaltfläche **Register** auf – zum Beispiel, wenn das Kennwort nicht mindestens ein nicht alphanumerisches Zeichen enthält oder wenn das Kennwort keinen Großbuchstaben aufweist (siehe Bild 8).

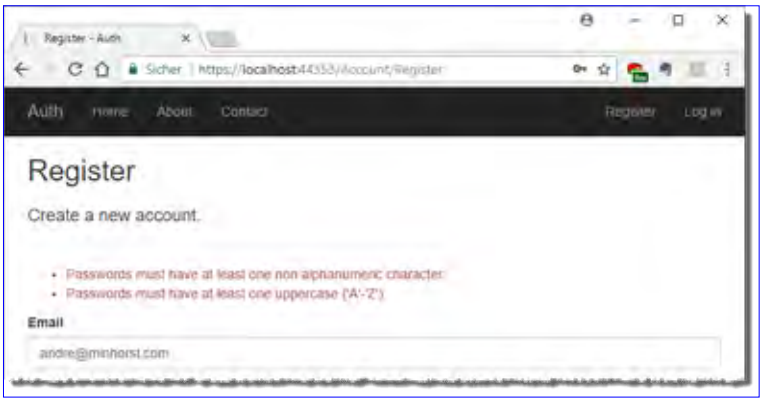


Bild 8: Fehlgeschlagene Registrierung

Fehler beim Registrieren ohne Datenbank

Wenn wir es schließlich geschafft haben, eine gültige E-Mail-Adresse und zwei den Anforderungen entsprechende und übereinstimmende Kennwörter einzugeben, folgt leider noch eine Fehlermeldung (siehe Bild 9). Diese weist uns darauf hin, dass eine Datenbank mit einem recht kryptischen Namen nicht geöffnet werden konnte. Zum Glück liefert die Meldung auch gleich noch einen Hinweis, wie wir die fehlende Datenbank bereitstellen können.

Bild 9: Fehler nach dem Absenden der Registrierung

Verbindungszeichenfolge anpassen und Datenbank erstellen

Dies holen wir nun nach, allerdings erst nach dem Anpassen der Verbindungszeichenfolge. Diese enthält nämlich auch den Namen der zu verwendenden Datenbank. Die Verbindungszeichenfolge finden wir in der Datei **appsettings.json**, die aktuell wie in Bild 10 aussieht. Hier können wir, bevor die Datenbank tatsächlich erstellen, noch den Namen der Datenbank auf eine eingängigere Bezeichnung einstellen, zum Beispiel **DbAuth**.

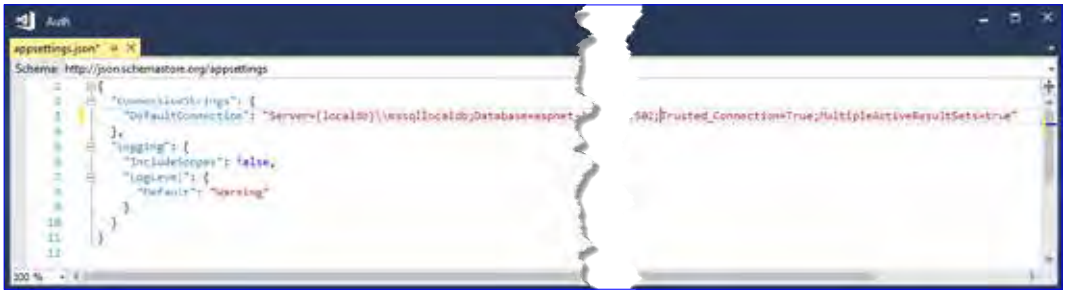


Bild 10: Ändern des Datenbanknamens in der Verbindungszeichenfolge

Danach erstellen wir die Datenbank. Dazu öffnen wir mit dem Menü-Eintrag **Extras|NuGet-Paket-Manager|Paket-Mana-**

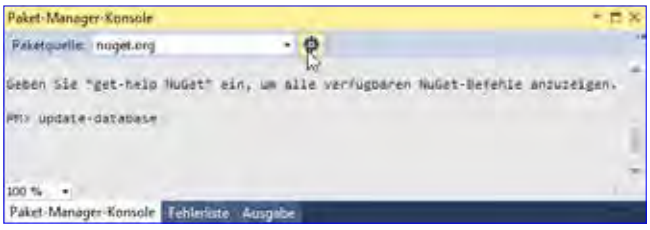


Bild 11: Starten der Erstellung der Datenbank

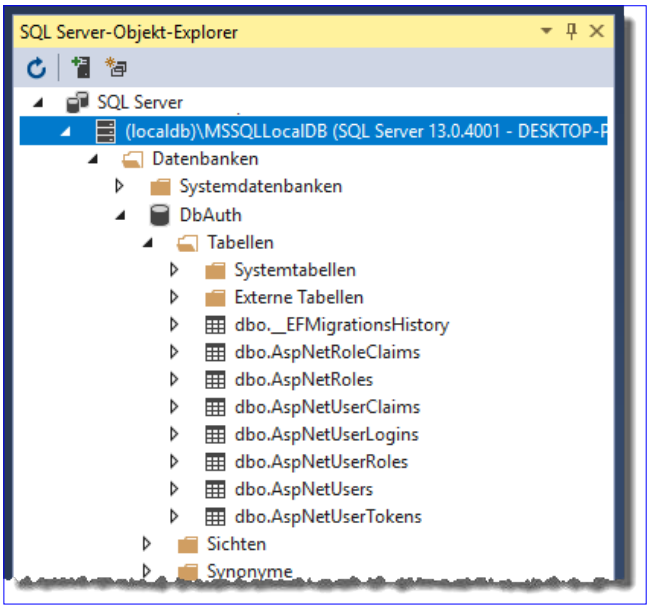


Bild 12: Die neue Datenbank **DbAuth**

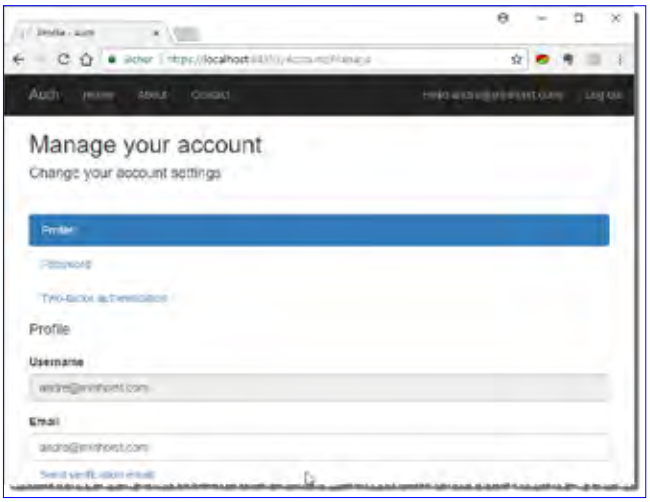


Bild 13: Die Registrierung hat funktioniert!

ger-Konsole den Bereich **Paket-Manager-Konsole**. Hier geben wir nun schlicht und einfach den Befehl **Update-Database** ein (siehe Bild 11). Dies startet die Erstellung der Datenbank mit dem in der Verbindungszeichenfolge angegebenen Namen **DbAuth**.

Nun wollen wir uns vergewissern, ob die Datenbank angelegt wurde und vor allem wie die Tabellen der Datenbank aussehen. Dazu starten wir mit **Ansicht|SQL Server-Objekt-Explorer** die gleichnamige Ansicht in Visual Studio 2017. Öffnen wir hier die Einträge **DbAuth** und darunter Tabellen, sehen wir, dass die Datenbank nicht nur angelegt wurde, sondern auch noch einige Tabellen enthält. Diese heißen etwa **AspNetUsers** oder **AspNetUserTokens** (siehe Bild 12).

Neuer Registrierungsversuch

Damit können wir nun erneut probieren, einen Benutzer zu registrieren. Dazu starten wir die zwischenzeitlich beendete Webanwendung erneut und klicken im nun erscheinenden Browser-Fenster auf den Link **Register**.

Diesmal gelingt die Registrierung! Nach dem Mausklick auf die Schaltfläche **Register** wird die Seite aktualisiert und zeigt im oberen Bereich den Eintrag **Hello andre@minhorst.com!** an. Ein Klick auf diesen Eintrag liefert die Profil-Informationen zum neu erstellten Benutzerkonto (siehe Bild 13).

Wenn wir uns nun mit einem Klick auf den Link **Log out** abmelden, können wir uns anschließend per Klick auf **Log in** erneut anmelden.

Ein Blick in die Tabelle **dbo.AspNetUsers**, die wir mit dem Kontextmenü-Eintrag **Daten anzeigen** des Elements

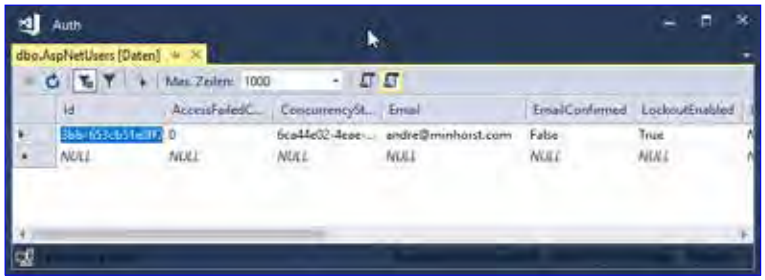


Bild 14: Gespeicherter Benutzer-Datensatz

DatenbankenIDbAuthITabellenIdbo.AspNetUsers öffnen, zeigt einen neuen Datensatz mit der von uns angegebenen E-Mail-Adresse (siehe Bild 14).

Änderungen durch die Benutzerverwaltung

Durch das Auswählen der Benutzerverwaltung beim Anlegen des Projekts wurden noch einige weitere Elemente zum Projekt hinzugefügt. Da wären zunächst einige Anweisungen, die der Datei **Startup.cs** hinzugefügt wurden und die in der Methode Konstruktor-Methode **Startup** dafür sorgen, dass alle für die Benutzerverwaltung notwendigen Klassen initialisiert werden. Dazu gehören auch die Elemente, die für den Zugriff auf die Datenbank notwendig sind und die Sie schon aus den Artikeln kennen, die sich mit der Programmierung von Desktop-Anwendungen auf Basis des Datenbankzugriffs mittels Entity Framework beschäftigen. Wichtig ist auch, dass die Methode **Configure** nun eine Anweisung enthält, welche die Authentifizierung aktiviert:

```
app.UseAuthentication();
```

Die Datei **appsettings.json**, welche die Anwendungseinstellungen aufnimmt, wurde um die Verbindungszeichenfolge erweitert, die wir ja schon geändert haben.

Bestätigen eines Kontos

Wenn der Benutzer ein neues Konto anlegt, erstellt die Anwendung wie oben gezeigt einen neuen Datensatz in der Tabelle **dbo.AspNetUsers**. Wenn Sie diesen Datensatz im Detail ansehen, finden Sie im Feld **EmailConfirmed** den Wert **False**. Das bedeutet, dass das Benutzerkonto noch nicht bestätigt wurde. Heutzutage ist dies allerdings ein wichtiger Bestandteil, um sicherzustellen, dass das Konto auch tatsächlich durch den Benutzer erstellt wurde, dessen E-Mail-Adresse bei der Anmeldung verwendet wurde. Erst wenn dieser Benutzer eine E-Mail an seine Adresse geschickt bekommt und einen darin enthaltenen Bestätigungslink anklickt, wird das Feld **EMailConfirmed** auf den Wert **True** eingestellt. Dieses Verfahren nennt sich Double-Opt-In und stellt sicher, dass niemand anderes ein Konto unter Ihrer E-Mail-Adresse anlegen kann.

Als Erstes ändern wir nun eine Einstellung, die wir in der Methode **ConfigureServices** der Klasse **Startup.cs** zuweisen. Dabei legen wir für den Aufruf der **AddIdentity**-Methode den Wert von **config.SignIn.RequireConfirmedEMail** auf **true** fest (siehe ab Markierung **//Änderung Start**):

```
public void ConfigureServices(IServiceCollection services) {
    services.AddDbContext<ApplicationDbContext>(options =>
        options.UseSqlServer(Configuration.GetConnectionString("DefaultConnection")));
    //Änderung Start:
    services.AddIdentity<ApplicationUser, IdentityRole>(config => {
        config.SignIn.RequireConfirmedEmail = true;
    })
    //Änderung Ende
        .AddEntityFrameworkStores<ApplicationDbContext>()
        .AddDefaultTokenProviders();
    services.AddMvc()
        .AddRazorPagesOptions(options => {
```

ASP.NET Core: Validierung

Genau wie in Desktop-Anwendung ist auch in Webanwendungen die Validierung der Benutzereingaben ein wichtiges Feature. Unter ASP.NET Core gibt es für die Razor Pages erfreulicherweise eine Standardvorgehensweise, die wir in diesem Artikel vorstellen werden.

Vorbereitung

Für das Nachvollziehen der Beispiele in diesem Artikel legen Sie in Visual Studio 2017 ein neues Projekt namens Validierung an. Es wird mit der Projektvorlage **Visual C#|Web|ASP.NET Core Webanwendung** erstellt. Im zweiten Vorlagendialog wählen Sie dann den Typ **Webanwendung** aus. Wir wollen die Seite **Pages/Customers/Create.cshtml** nutzen, um unsere Beispiele auszuprobieren. Diese wollen wir über einen neuen Menüeintrag der Webanwendung aufrufen, den wir in der Datei **_Layout.cs** einfügen, und zwar an dieser Stelle:

```
...
<div class="navbar-collapse collapse">
    <ul class="nav navbar-nav">
        <li><a asp-page="/Index">Home</a></li>
        <li><a asp-page="/About">About</a></li>
        <li><a asp-page="/Customers/
Create">Neuer Kunde</a></li>
    </ul>
</div>
...
```

Insgesamt werden wir neben Änderung der Datei **_Layout.cs** noch zwei Dateien samt Verzeichnissen hinzufügen, nämlich **Models/Customer.cs** und **Pages/Customers/Create.cshtml** (siehe Bild 1).

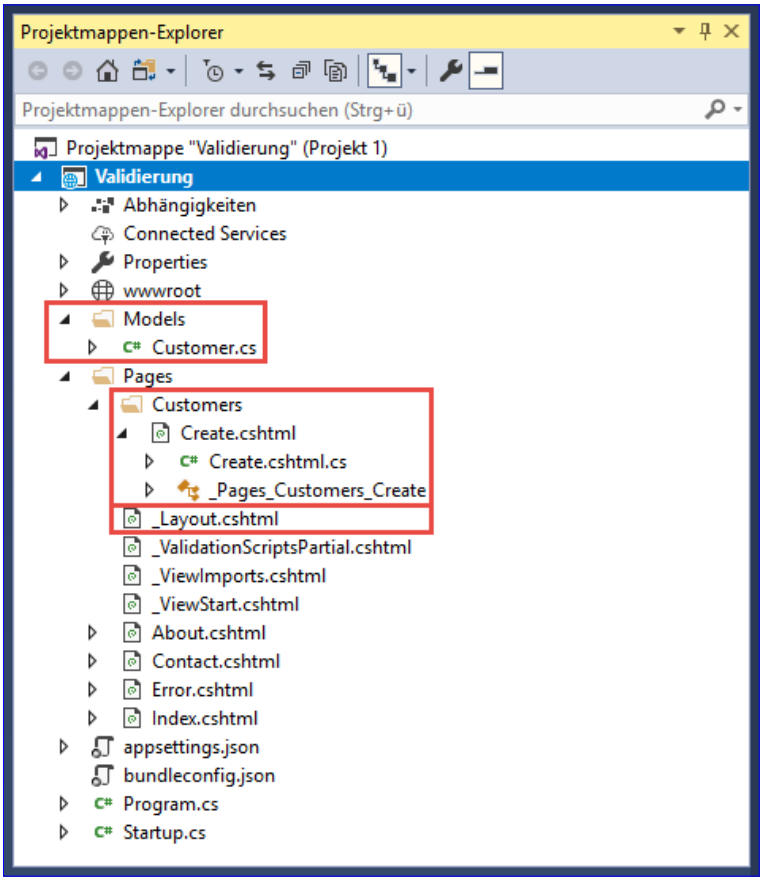


Bild 1: Anlage einiger Verzeichnisse und Dateien

Grundlagen

In Webanwendungen landen die Daten aus Tabellen zunächst in einem Objekt auf Basis einer Klasse, bevor diese auf einer Webseite dargestellt werden – diese Vorgehensweise kennen Sie ja auch schon von den Artikeln über den Einsatz des Entity Frameworks in Desktop-Anwendungen. Der Vorteil dieser Vorgehensweise ist, dass die Validierungsregeln nur an einer Stelle angelegt werden müssen und nicht etwa auf jeder Seite, welche die Eingabe oder Bearbeitung der Daten der betroffe-

nen Klasse erlauben soll. Die Validierungsregeln werden hier wie da direkt in den Klassen angelegt, welche die Tabellen der Datenbank repräsentieren. Dazu legen wir zu Beispielzwecken diesmal keine Klasse auf Basis einer Tabelle an, sondern eine einfache Klasse ohne Bindung an eine Datenbank.

Bild 2: Hinzufügen einer Klasse zum Verwalten von Kunden

Zu Beispielzwecken legen wir nun eine Klasse namens **Customers.cs** an, und zwar in einem zuvor erstellten Verzeichnis namens Models direkt unterhalb des Projektordners. Dort wählen Sie dann den Kontextmenü-Eintrag **Hinzufügen|Klasse...** aus und geben im nun erscheinenden Dialog **Neues Element hinzufügen** den Namen für die Klasse ein – hier **Customer.cs** (siehe Bild 2).

Die Klassendatei mit der Klasse **Customer** ergänzen wir nun wie in folgendem Code um einige Beispielfelder. Wozu das Feld namens **Rating** dient? Es hat einen bestimmten Zweck – außer als Beispiel für ein **Decimal**-Feld ...

```
public class Customer {
    public int ID { get; set; }
    public string FirstName { get; set; }
    public string LastName { get; set; }
    public string Street { get; set; }
    public string Zip { get; set; }
    public string City { get; set; }
    public string Email { get; set; }
    public string Url{ get; set; }
    public string Phone { get; set; }
    public DateTime Birthday { get; set; }
    public decimal Rating { get; set; }
}
```

Wenn wir nun ein Formular auf Basis dieser Daten anlegen, sieht dies wie in gekürzter Form wie folgt aus:

```
@page
@model Validierung.Pages.Customers.CreateModel
```

```
@{ ViewData["Title"] = "Create"; }
<h2>Datensatz erstellen</h2>
<div class="row">
    <div class="col-md-4">
        <form method="post">
            <div asp-validation-summary="ModelOnly" class="text-danger"></div>
            <div class="form-group">
                <label class="control-label">Vorname:</label>
                <input asp-for="Customer.FirstName" class="form-control" />
                <span asp-validation-for="Customer.FirstName" class="text-danger"></span>
            </div>
            ...
            <div class="form-group">
                <input type="submit" value="Create" class="btn btn-default" />
            </div>
        </form>
    </div>
</div>
```

Wir sehen hier im oberen Teil die Razor-Anweisungen, mit denen unter anderem die Klasse **Validierung.Pages.Customers.CreateModel** als Modell referenziert wird. Darunter folgen einige Elemente samt **form**-Element. Die für die Validierung wichtigen Elemente sind das Element mit dem Attribut **asp-validation-summary** über den übrigen Elementen sowie das Element mit dem Attribut **asp-validation-for** für jedes einzelne Steuerelement, das an eines der Felder der Modell-Klasse gebunden ist.

Bevor wir auf diese eingehen, schauen wir uns noch an, wie wir die Klasse mit ein paar Validierungen ausstatten und wie das **PageModel** und das Model für die Seite aussehen.

Model der HTML-Seite

Die Code behind-Datei enthält den folgenden Code, unter anderem mit dem **PageModel**:

```
... weitere Namespace-Referenzen
using Validierung.Models;

namespace Validierung.Pages.Customers {
    public class CreateModel : PageModel {
        public IActionResult OnGet() {
            return Page();
        }
        [BindProperty]
        public Customer Customer { get; set; }
        public IActionResult OnPost() {
```

```
            if (!ModelState.IsValid) {
                return Page();
            }
            return RedirectToPage("./Index");
        }
    }
}
```

Hier definieren wir einen Verweis auf das Verzeichnis Entitäten, also **Validierung.Models**. Dann fügen wir eine **OnGet**-Methode ein, welche beim Aufruf die auf Basis von **Create.cshtml** zusammengestellt Seite zurückliefert. Wir binden die Seite an die Klasse **Customer** als Model. Dazu versehen wir die öffentliche Variable **Customer** des Typs **Customer** mit dem Attribut **[BindProperty]**. Interessant ist die Methode **OnPost**, die letztlich die Validierung ausführt – und zwar durch die Methode **IsValid** des Objekts **ModelState**. Liefert dieses nicht das Ergebnis **True**, wird wieder die Seite mit dem zuvor ausgefüllten Formular zurückgeliefert. Erst wenn **IsValid** den Wert **True** liefert, zeigt die Webanwendung die dann vorgesehene Seite an – in diesem Fall die **Index**-Seite. Das Objekt **ModelState** ist übrigens ein Element des Namespaces **Microsoft.AspNetCore.MVC**. Wenn die Methode **IsValid** von **ModelState** aufgerufen wird, untersucht die Webanwendung, ob die im Model **Customer** festgelegten Validierungsregeln erfüllt sind. Diese müssen wir allerdings erst noch hinzufügen.

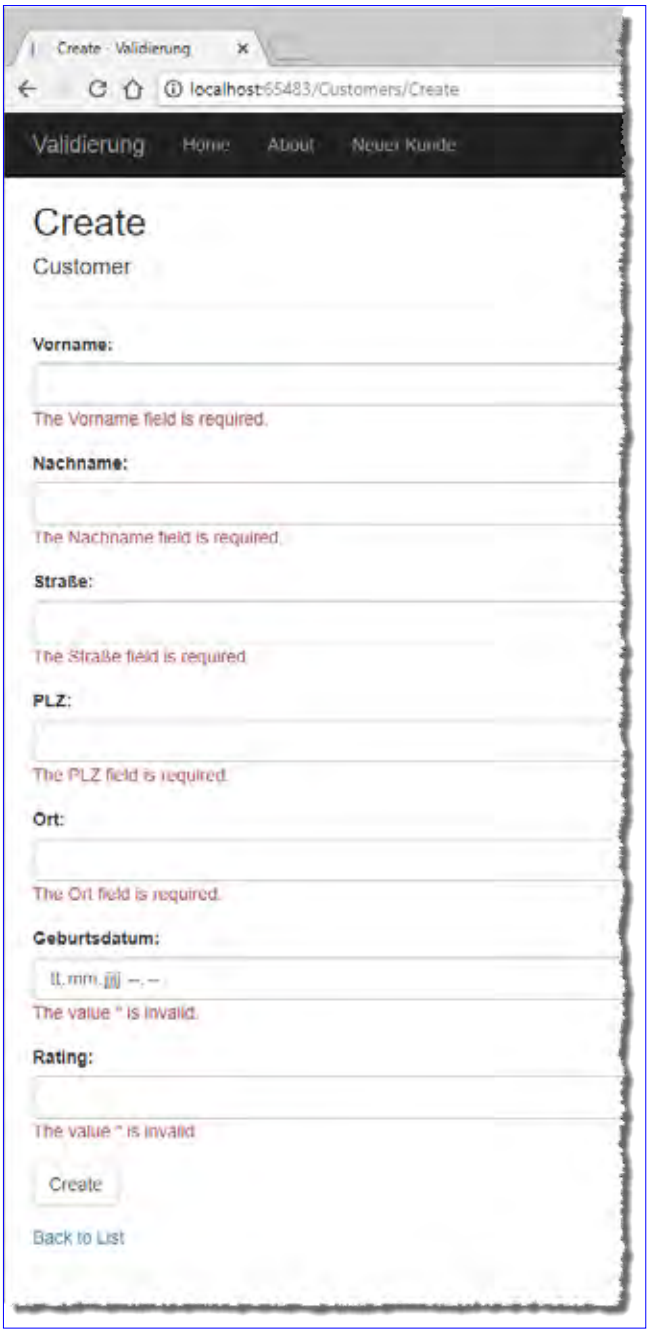


Bild 3: Validierungsmeldungen

Validierungsregeln in das Model einfügen

Um dem Model, also der Klasse **Customer.cs**, die Validierungsregeln hinzuzufügen, müssen wir zunächst den Namespace **System.ComponentModel.DataAnnotations** hinzufügen:

```
using System.ComponentModel.DataAnnotations;
```

Validierungsregel für Pflichtangaben

Die meistgenutzte Validierungsregel dürfte die sein, die prüft, ob erforderliche Werte angegeben wurden. Um ein Feld in der Klasse **Customer** mit dieser Validierungsregel zu belegen, fügen wir vor dem jeweiligen Feld die Data Annotation namens **Requery** ein – hier an einigen Beispielen zu sehen:

```
public class Customer {
    public int ID { get; set; }
    [Required]
    public string FirstName { get; set; }
    [Required]
    public string LastName { get; set; }
    [Required]
    public string Street { get; set; }
    ...
}
```

Wenn wir die Anwendung nun starten, über den Menüeintrag **Neuer Kunde** zur Seite **Create.cshtml** wechseln und dort direkt auf die Schaltfläche mit der Beschriftung **Create** klicken, erhalten wir die Validierungsmeldungen aus Bild 3. Es klappt also alles wie gewünscht. Nun schauen wir uns weitere Möglichkeiten an.

Tag Helper für die Validierung

Bei **asp-validation-summary** und **asp-validation-for** handelt es sich um sogenannte Tag-Helper (mehr Tag-Helper stellen wir im Artikel **ASP.NET Core: Tag Helper** vor). Der Tag Helper **asp-validation-summary** wird in der Regel einmal oberhalb eines Formulars angegeben. Dort hat er die Aufgabe, eventuell auftretende Validierungshinweise auszugeben, die nicht nur ein spezielles Feld betreffen, sondern mehrere Felder oder allgemeine Hinweise. Für dieses Attribut gibt es drei mögliche Werte:

- **All**
- **ModelOnly**
- **None**

Für **asp-validation-summary** ist aktuell der Wert **ModelOnly** festgelegt. Wir könnten diesen Wert einmal ändern in **All** und schauen uns an, was geschieht, wenn wir das Formular erneut ohne Füllen eines der Felder absenden (siehe Bild 4):

```
<div asp-validation-summary="All" class="text-danger"></
div>
```

Es erscheint also schon vor dem Feld **Vorname** eine Übersicht aller fehlerhaft ausgefüllten Eingabefelder. Das ist ehrlich gesagt nicht notwendig, wenn schon jedes einzelne Feld mit einem entsprechenden Text ausgestattet wird. Leider sind

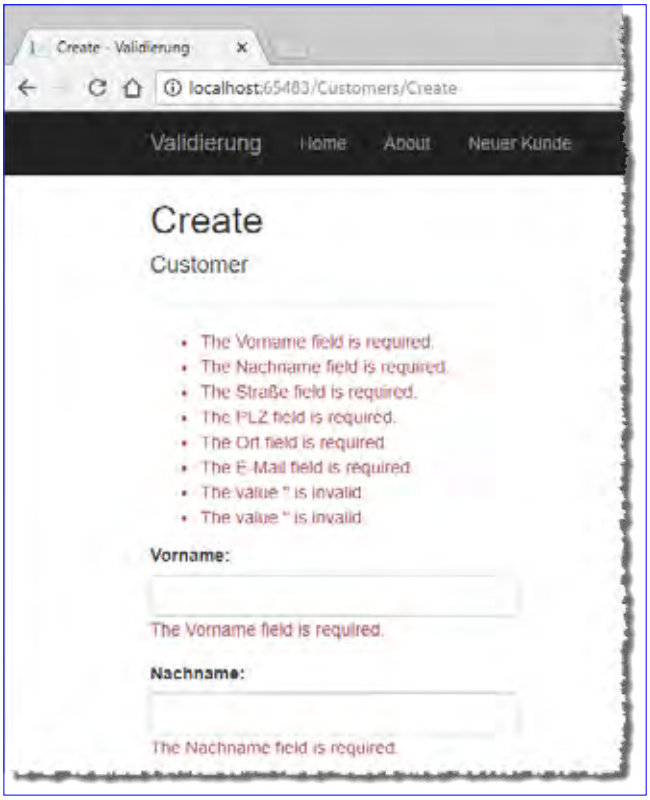


Bild 4: Validierungsmeldungen zusammengefasst im Kopfbereich

diese Texte auch noch in englischer Sprache – wir schauen uns also gleich an, wie Sie hier die deutschen Texte angeben können.

Wozu aber dient dann die Einstellung **ModelOnly**? Auch dieses hat seine Berechtigung, warum, sehen wir uns ebenfalls weiter unten an.

Validierungstexte in Deutsch

Die Validierungstexte können Sie direkt in den DataAnnotation-Elementen für die einzelnen Felder ergänzen, und zwar für das Element **Required**. Diesem fügen Sie in Klammern noch den Parameter **ErrorMessage** mit dem gewünschten Text hinzu.

Die Meldungen werden dann sowohl im Kontext mit den einzelnen Textfeldern als auch in der Übersicht der Validierungsmeldungen in der gewünschten Form ausgegeben (siehe Bild 5).

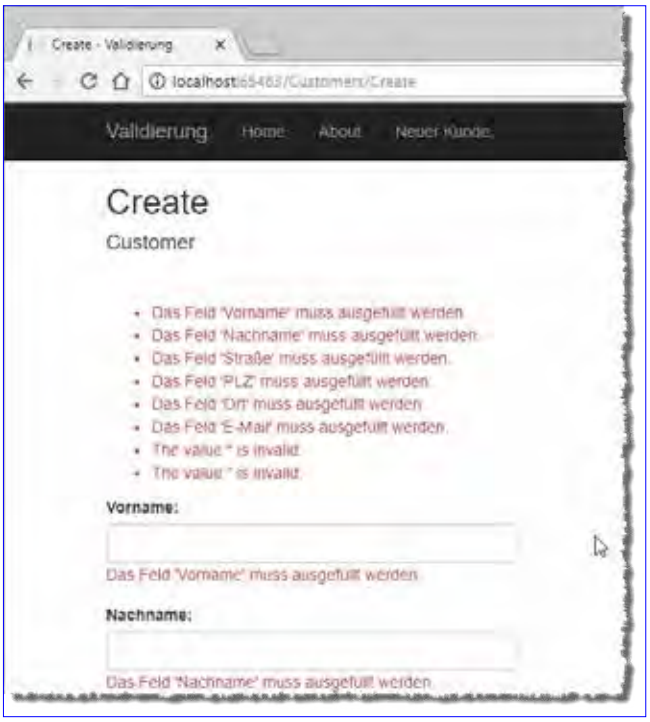


Bild 5: Deutsche Validierungsmeldungen

Das sieht dann etwa wie folgt aus:

```
public class Customer {
    public int ID { get; set; }
    [Required(ErrorMessage = "Das Feld 'Vorname' muss ausgefüllt werden.")]
    public string FirstName { get; set; }
    [Required(ErrorMessage = "Das Feld 'Nachname' muss ausgefüllt werden.")]
    public string LastName { get; set; }
    [Required(ErrorMessage = "Das Feld 'Straße' muss ausgefüllt werden.")]
    public string Street { get; set; }
    ...
}
```

Allerding sehen wir im Screenshot ganz unten in der Liste der Validierungsfehler noch zwei Mal den Inhalt **The value '' is invalid**. Was bedeutet dies? Es gibt hier anscheinend Validierungsregeln, die unsere angegebene Regel namens **Required** außer Kraft setzen. Das ist bei dem Datumsfeld **Geburtsdatum** und beim **Decimal**-Feld **Rating** der Fall.

Weitere Validierungsmöglichkeiten

Nachfolgend schauen wir uns die folgenden Validierungsmöglichkeiten an:

- **MaxLength(<Länge>)**: Die Zeichenkette darf maximal **<Länge>** Zeichen lang sein.

ASP.NET Core: Anwendung veröffentlichen

Die ganzen Themen rund um die Entwicklung von Webanwendungen sind natürlich nutzlos, wenn Sie die Anwendung dann über das Internet nutzen können. Also schauen wir uns in diesem Beitrag an, wie wir unsere mit Datenbankanbindung versehene Webanwendung aus dem Artikel »Razor Pages mit Datenbankanbindung« ins Internet bringen.

Voraussetzungen

Um eine ASP.NET Core-Webseite online zu stellen, benötigen Sie zunächst ein Microsoft Azure-Konto. Dieses können Sie hier anlegen:

`https://azure.microsoft.com/de-de/free/`

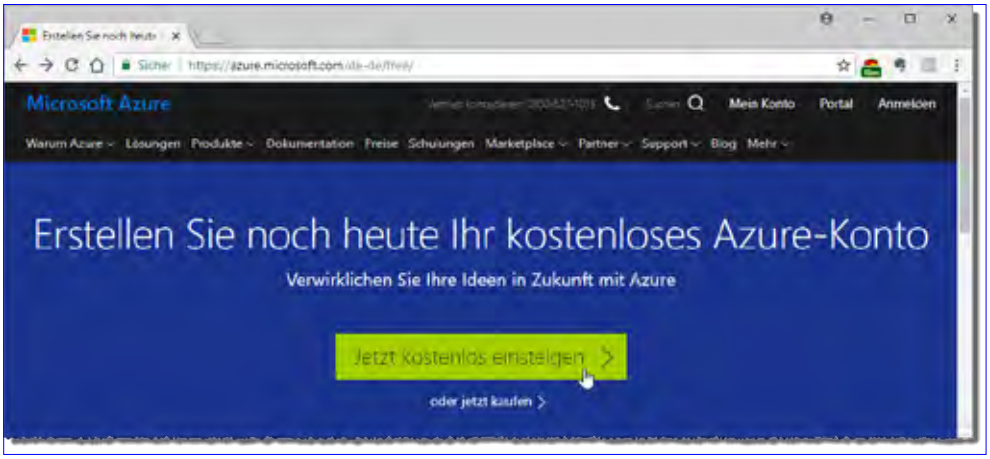


Bild 1: Erstellen eines Azure-Kontos

Dazu klicken Sie auf dieser Seite auf den Link **Jetzt kostenlos einsteigen** (siehe Bild 1). Danach melden Sie sich zunächst mit Ihrem Windows-Konto an, das Sie ja auch schon zur Installation von Visual Studio 2017 verwendet haben dürften.

Veröffentlichen des Projekts

Nachdem diese Schritte abgeschlossen sind, klicken Sie im Projektmappen-Explorer auf den Kontextmenü-Eintrag **Veröffentlichen...** des Projekts (siehe Bild 2).

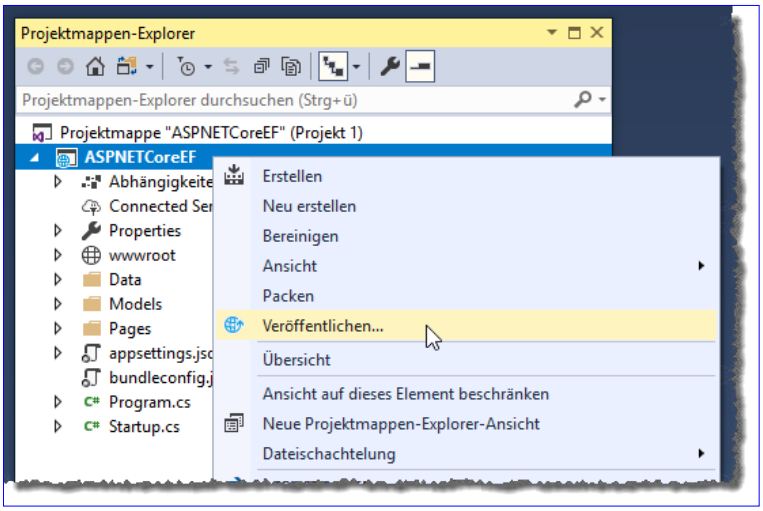


Bild 2: Veröffentlichen einer Webanwendung

Es erscheint der Dialog aus Bild 3, in dem Sie als Veröffentlichungsziel den Eintrag **App Service beibehalten** und rechts unter **Azure App Service** die Option **Neues Element erstellen** übernehmen. Dann klicken Sie auf die Schaltfläche **Veröffentlichen**.

Der folgende Dialog fragt, ob Sie ein kostenloses Azure Konto erstellen oder ein vorhandenes Konto nutzen wollen (siehe Bild 4).

Standardmäßig haben Sie, auch wenn sie bereits ein Windows-Konto besitzen, über welches Sie

auch Visual Studio 2017 heruntergeladen haben, noch kein Azure-Konto. Also legen wir eines über den Link **Kostenloses Azure-Konto erstellen** an.

Hier wählen Sie das bestehende Microsoft-Konto aus oder legen ein neues an und erstellen dann darauf basierend ein neues Azure-Konto. Hier erfolgen einige Sicherheitsabfragen, SMS, Eingaben von Sicherheitscodes et cetera bis Sie auf Ihr Konto zugreifen können.

Außerdem müssen Sie eine Kreditkartennummer angeben. Nach vollständiger Einrichtung werden Sie schließlich auf die Übersichtsseite Ihres neuen Azure-Kontos weitergeleitet (siehe Bild 5).

Damit sind die Arbeiten an dieser Stelle beendet und wir kehren wieder zu Visual Studio zurück.

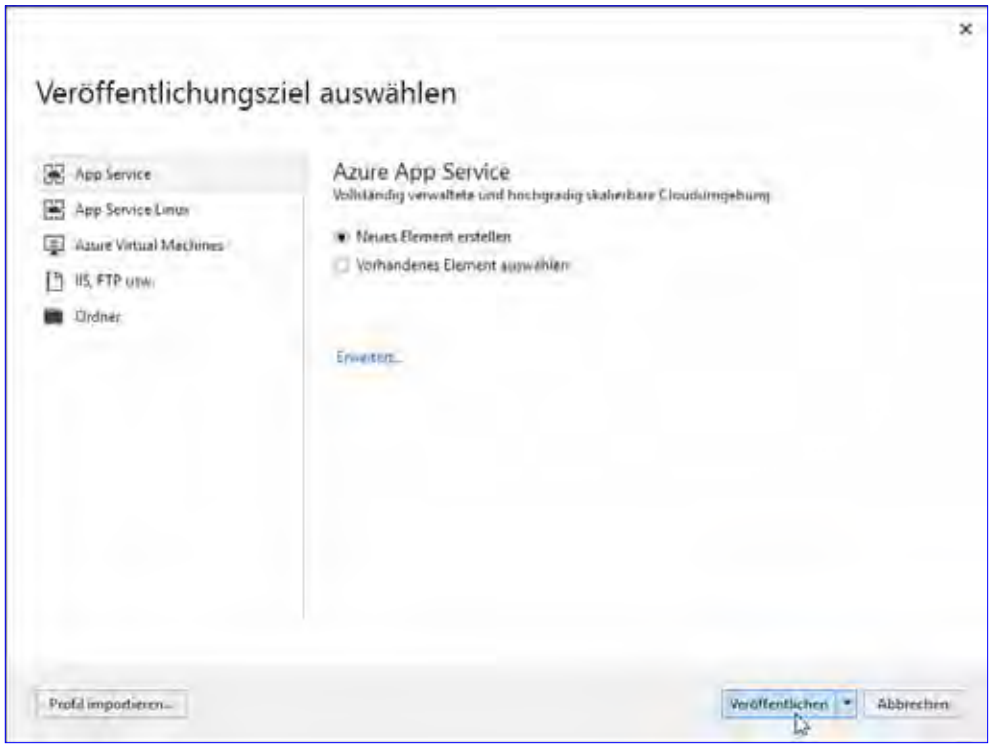


Bild 3: Veröffentlichungsziel auswählen

Bild 4: Azure-Konto erstellen oder an bestehendes Konto anmelden

App Service erstellen

Hier geht es nun mit dem Dialog aus Bild 6 weiter. Dieser zeigt automatisch Voreinstellungen für den neu zu erstellenden App-Service an, sobald Sie oben rechts ein Microsoft-Konto ausgewählt haben, das auch mit einem Azure-Konto verknüpft ist. Möglicherweise müssen Sie das Konto nochmals auswählen, damit dies geschieht. Hier sehen Sie nun einige Voreinstellungen, die wir zur Vereinfachung etwas anpassen:

- **App-Name**: Hier entfernen wir die Zahlen, damit wir einen griffigeren App-Namen erhalten.
- **Subscription**: Diesen Parameter stellen wir gleich über einen weiteren Parameter ein.

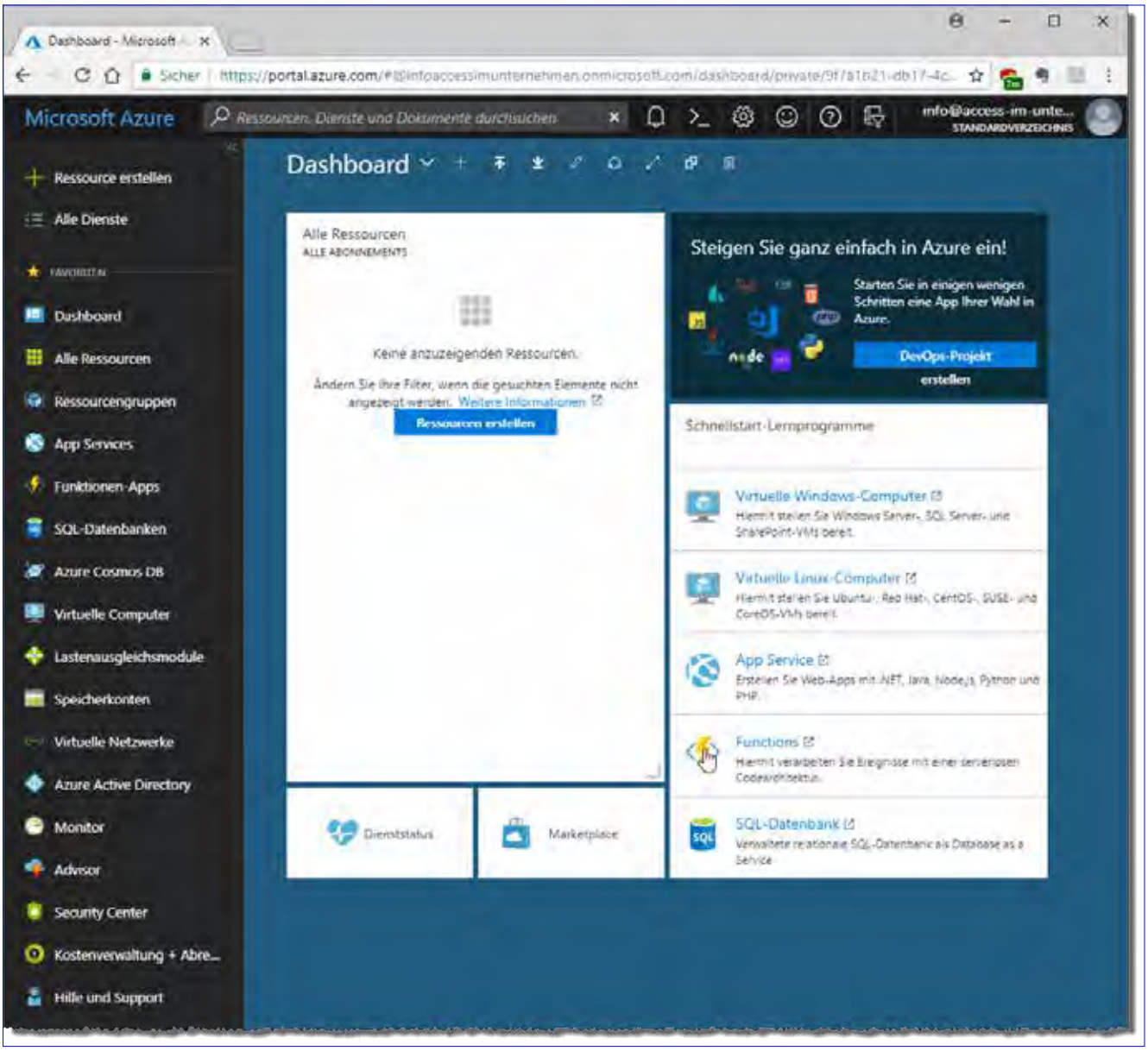


Bild 5: Ihr neues Azure-Konto

- **Ressourcengruppe**: Klicken Sie hier auf **Neu...**, um den Namen der Ressourcengruppe anzupassen.

- **Hostingplan**: Klicken Sie hier auf **Neu...**, erscheint der Dialog aus Bild 7. Hier geben Sie ebenfalls einen griffigeren Namen für das Feld **App Service Plan** ein, legen den Speicherort auf **West Europe** fest und wählen unter Größe vorsichtshalber den Eintrag **Kostenlos** aus. Dies passt dann auch den Wert des Parameters **Subscription** an.

Datenbank hinzufügen

Für diese Anwendung benötigen wir außerdem eine Datenbank auf dem Internetserver. Deshalb klicken Sie nun unter **Weitere Azure-Services durchsuchen** auf den Eintrag **SQL-Datenbank erstellen**. Wir haben aber noch keinen SQL Server angelegt.

Im nun erscheinenden Dialog **SQL-Datenbank** konfigurieren finden Sie daher neben der Einstellung **SQL Server** einen Link namens **Neu...**, den Sie nun betätigen.

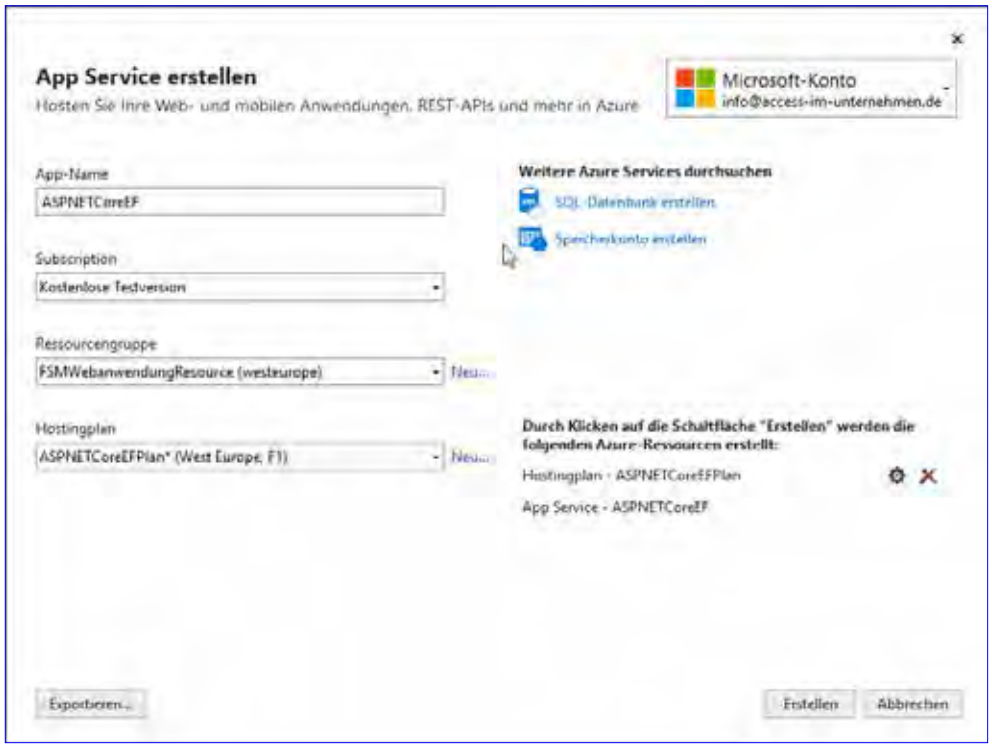
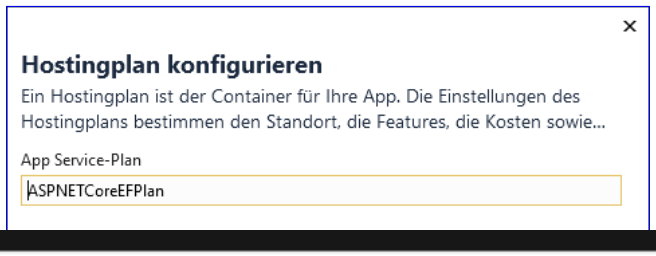
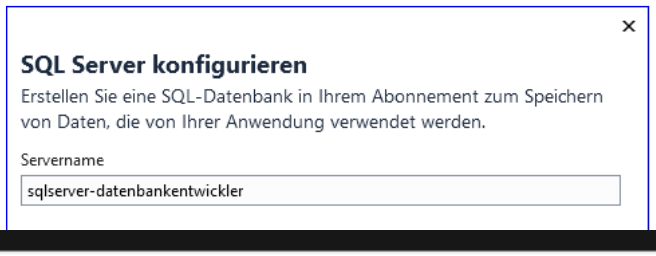


Bild 6: Erstellen des App-Services

Es erscheint der Dialog aus Bild 8. Hier geben Sie den Namen der Datenbank, einen Administratornamen und das Kennwort ein.

Nach dem Betätigen der **OK**-Schaltfläche erscheint nochmal die Seite **SQL-Datenbank konfigurieren**, in dem der neu angelegt SQL Server ausgewählt ist (siehe Bild 9). Wenn Sie später weitere Datenbanken für Webanwendungen anlegen, können Sie den nun erstellten SQL Server erneut auswählen.





AUSGABE 04/2018

DATENBANK ENTWICKLER

MAGAZIN FÜR DIE DATENBANKENTWICKLUNG MIT VISUAL STUDIO FÜR DESKTOP, WEB UND CO.

TOP-THEMEN:

NEWS AND TOOLS	.NET Core auf Version 2.1 aktualisieren	SEITE 3
EF CORE	EDM: 1:n-Beziehungen per Code First	SEITE 5
EF CORE	EF Core: Klassendiagramm anzeigen	SEITE 21
ASP.NET CORE	Schnellstart mit Datenbank	SEITE 23
ASP.NET CORE	Razor-Pages: Daten aus Lookup-Tabellen	SEITE 55

André Minhorst Verlag

.NET Core auf Version 2.1 aktualisieren

Der Vorteil eines Access-Entwicklers gegenüber einem .NET-Entwickler ist, dass er sich in den letzten zehn Jahren kaum Sorgen über Updates machen muss. Bei .NET und auch .NET Core ist das anders: Hier gibt es alle paar Monate Neuigkeiten. In diesem Fall steht ein Update auf .NET Core 2.1 an. Dieser kurze Artikel zeigt, welche Schritte notwendig sind, um die in dieser Ausgabe vorgestellten Artikel nachvollziehen zu können.

.NET Core ist noch sehr frisch, daher liefert Microsoft in sehr kurzen Abständen neue Versionen. Diese neuen Versionen enthalten oft tolle Features, die wir auch in diesem Magazin vorstellen wollen.

Daher informieren wir Sie hier über Updates, die notwendig sind, damit die in den Artikel beschriebenen Beispiele lauffähig sind und nachvollzogen werden können.

In diesem Fall geht es um die neue Version von .NET Core mit der Nummer 2.1. Diese installieren Sie unter Visual Studio 2017 (in unserem Beispiel mit der Community Edition).

Den Download finden Sie zum Zeitpunkt der Erstellung dieses Artikels unter der folgenden Internetadresse (siehe Bild 1):

```
https://www.microsoft.com/net/download
```

Bild 1: Download von .NET Core SDK 2.1

Hier wählen Sie die Version mit dem SDK aus. Nach dem Download installieren Sie die neue Version durch einen Doppelklick auf die Datei **dotnet-sdk-2.1.302-win-x64.exe** (siehe Bild 2).

Gegebenenfalls bittet das Setup Sie darum, geöffnete Instanzen von Visual Studio zu schließen – damit sparen Sie sich einen späteren Neustart des Systems. Nach der Installation sind Sie gut auf die in dieser Ausgabe vorgestellten Artikeln rund um das Thema Webentwicklung vorbereitet!

Bild 2: Installation von .NET Core SDK 2.1

EDM: 1:n-Beziehungen per Code First

Im Artikel »EDM: Der Code First-Ansatz« haben wir gezeigt, wie Sie unter Nutzung des Entity Frameworks ein Datenmodell per Klassenmodell programmieren und dann mit ein paar Befehlen dafür sorgen, dass die Klassen in Form eines Datenmodells in einer Datenbank umgesetzt werden. Im vorliegenden Artikel schauen wir uns nun im Detail an, wie Sie 1:n-Beziehungen mit Code First definieren, die dann in die Zieldatenbank übertragen werden.

Vorbereitungen

Für diesen Artikel nutzen wir ein Projekt auf Basis der Vorlage **Visual C#|Windows Desktop|WPF-App** namens **CustomerManagement**. Die Techniken lassen sich aber auch in Projekten auf Basis anderer Vorlagen nutzen, beispielsweise in Webanwendungen.

Fügen Sie dem Projekt ein ADO.NET Entity Data Model hinzu.

- Dazu betätigen Sie den Kontextmenü-Eintrag **Hinzufügen|Neues Element** des Projekt-Eintrags im Projektmappen-Explorer.
- Im nun erscheinenden Dialog **Neues Element hinzufügen** wählen Sie den Eintrag **ADO.NET Entity Data Model** aus. Das Element benennen wir mit **CustomerManagementContext**.
- Wählen Sie im folgenden Dialog die Option **Leeres Code First-Modell** aus.

Damit erstellt der Assistent nun eine neue Klasse namens **CustomerManagementContext.cs**, die ein paar auskommentierte Beispielanweisungen enthält. Außerdem finden Sie in der Datei **App.conf** ein Element namens **connectionStrings**. Hier passen wir den Namen der zu erstellenden Datenbank mit dem Attribut **initial catalog** auf **CustomerManagement** an:

```
01.     <connectionStrings>
02.       <add name="CustomerManagementContext" connectionString="data source=(LocalDb)\MSSQLLocalDB;
              initial catalog=CustomerManagement;integrated security=True;MultipleActiveResultSets=True;
              App=EntityFramework" providerName="System.Data.SqlClient" />
03.     </connectionStrings>
```

Dadurch wird die Datenbank beim Erstellen **CustomerManagement** benannt.

Beispiel

Unser einfaches Beispiel soll eine Klasse namens **Customer** und eine namens **Salutation** enthalten und daraus die Tabellen **Customers** und **Salutations** erzeugen, die über ein Feld **SalutationID** der Tabelle **Customers** und das gleichnamige Feld der Tabelle **Salutations** miteinander verknüpft sind.

Klassen erstellen und DbContext einrichten

Zur Erinnerung: Wir können die einzelnen Klassen direkt in der Klassendatei **CustomerManagementContext.cs** unterbringen, aber auch in jeweils einer eigenen Datei. In letzterem Fall würden wir die Klassen aus Gründen der Übersicht in einem Unterverzeichnis namens **Model** anlegen. Dieses Verzeichnis fügen wir über den Kontextmenü-Eintrag **Hinzufügen|Neuer Ordner** zum Projektordner hinzu. Dann legen wir darin eine neue Klassendatei namens **Customer.cs** an. Dadurch wird für diese Klasse automatisch der Namespace **CustomerManagement.Models** festgelegt, was wir später noch berücksichtigen müssen. Wir ändern die Sichtbarkeit der Klasse auf öffentlich (**public**) und fügen ein paar Felder hinzu – unter anderem ein Feld namens **ID**, das später als Primärschlüsselfeld verwendet werden soll und eines namens **SalutationID**, das als Fremdschlüsselfeld zur Tabelle **Salutations** zum Einsatz kommt:

```
namespace CustomerManagement.Models {
    public class Customer {
        public int ID { get; set; }
        public string Company { get; set; }
        public string FirstName { get; set; }
        public string LastName { get; set; }
        public int SalutationID { get; set; }
    }
}
```

Auf die gleiche Weise erzeugen wir eine Klasse für die Anreden, die wir Salutation nennen. Auch diese legen wir im Verzeichnis **Models** an und stellen die Sichtbarkeit auf öffentlich ein. Dieses Klasse soll die Felder **ID** und **Name** erhalten:

```
namespace CustomerManagement.Models {
    public class Salutation {
        public int ID { get; set; }
        public string Name { get; set; }
    }
}
```

Nun müssen wir noch die **DbSet**-Elemente in der Klasse **CustomerManagementContext** hinzufügen. Damit wir dabei auf die Klassen **Customer** und **Salutation** als Datentypen der Auflistungen zugreifen können, machen wir den Namespace **CustomerManagement.Models** mit der **using**-Anweisung verfügbar:

```
namespace CustomerManagement {
    ...
    using CustomerManagement.Models;
```

Dann folgt die eigentliche Klasse mit dem zunächst leeren Konstruktor:

```
    public class CustomerManagementContext : DbContext {
```

```
        public CustomerManagementContext() : base("name=CustomerManagementContext") {
        }
```

Diese enthält dann schließlich die **DbSet**-Elemente für die **Customers**- und die **Salutations**-Auflistungen:

```
        public virtual DbSet<Salutation> Salutations { get; set; }
        public virtual DbSet<Customer> Customers { get; set; }
    }
}
```

Damit haben wir die Voraussetzungen geschaffen, erstmalig die Datenbank zu erstellen. Das erledigen wir initial mit den drei folgenden Anweisungen, die wir in der Paket-Manager-Konsole absetzen. Die erste lautet wie folgt und aktiviert die Migration zum Datenbankserver. Dabei wird ein Verzeichnis namens **Migrations** erstellt sowie eine Datei namens **Configurations.cs**, die unter anderem die **Seed**-Methode enthält, in der Sie Anweisungen zum Füllen der erzeugten Tabellen mit Daten hinzufügen können (dieser Befehl ist ab Version 2.1 nicht mehr nötig):

```
enable-migrations
```

Die zweite erstellt den Code für die initiale Migration. Dieser wird in der Datei angelegt, die mit einer Zahl beginnt und mit **...Init.cs** endet:

```
add-migration Init
```

Nun müssen Sie die Migration nur noch aufrufen, was Sie zum Beispiel mit der folgenden Anweisung erledigen:

```
update-database
```

Alternativ können Sie dafür sorgen, dass die Migrationen beim Start der Anwendung automatisch durchgeführt werden. Dazu rufen Sie einmalig den folgenden Befehl in der Paket-Manager-Konsole auf:

```
enable-migrations -EnableAutomaticMigration:$true
```

Sollten Sie die Migration bereits einmal ausgeführt haben, müssen Sie noch den Parameter **-Force** anhängen:

```
enable-migrations -EnableAutomaticMigration:$true -Force
```

Dies passt die Klasse **MigrationsIConfiguration.cs** an, in der die Eigenschaft **AutomaticMigrationsEnabled** in der Konstruktor-Klasse auf **true** gesetzt wird:

```
public Configuration() {
    AutomaticMigrationsEnabled = true;
```

```
}
```

In der Klasse **CustomerManagementContext** fügen wir der Konstruktor-Klasse die folgende Anweisung hinzu:

```
public class CustomerManagementContext : DbContext {
    public CustomerManagementContext() : base("name=CustomerManagementContext") {
        Database.SetInitializer(new MigrateDatabaseToLatestVersion<CustomerManagementContext, Migrations.Configuration>());
    }
    public virtual DbSet<Salutation> Salutations { get; set; }
    public virtual DbSet<Customer> Customers { get; set; }
}
```

Dabei ist das erste Element in spitzen Klammern hinter dem Objekt **MigrateDatabaseToLatestVersion** der Name der Kontextklasse und das zweite die **Configuration**-Klasse im Namespace **Migrations**.

Um die Migration beim Start der Anwendung jeweils auszulösen (sofern Änderungen vorliegen), erweitern Sie noch die Konstruktor-Methode der Klasse **MainWindow**:

```
public MainWindow() {
    InitializeComponent();
    CustomerManagementContext dbContext = new CustomerManagementContext();
    dbContext.Database.Initialize(true);
}
```

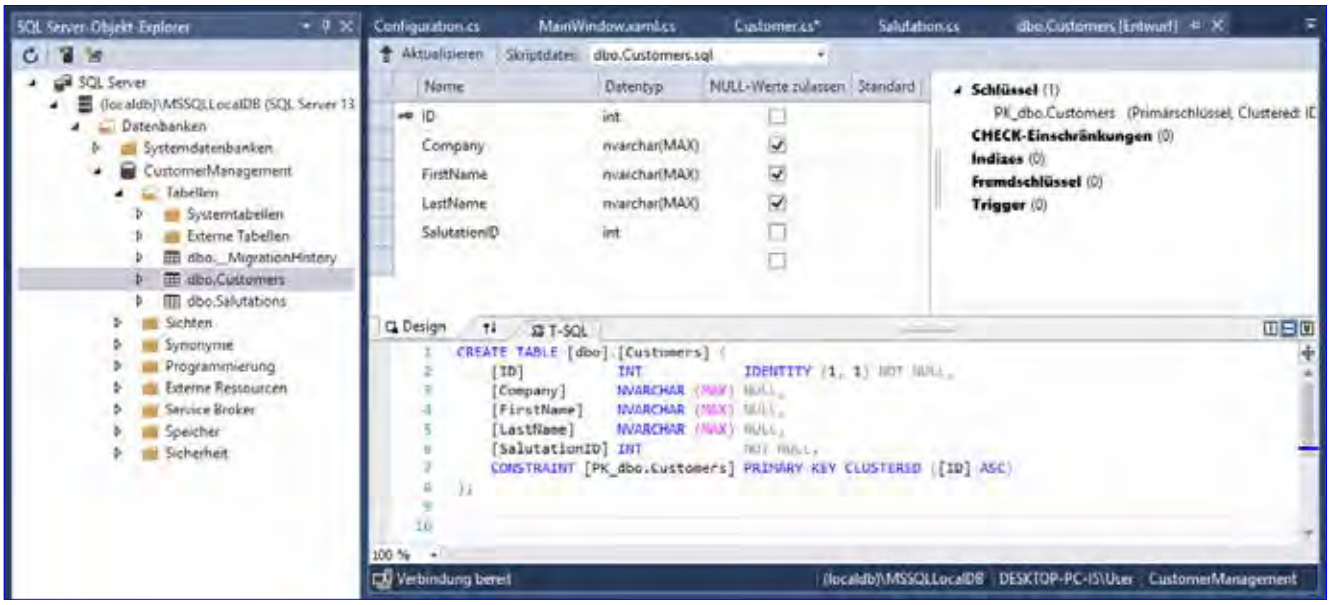


Bild 1: Definition der Tabelle **Customers**

EDM: 1:1-Beziehungen per Code First

Im Artikel »EDM: 1:n-Beziehungen per Code First« haben wir gezeigt, wie Sie unter Nutzung des Entity Frameworks zwei Klassen so gestalten, dass diese beim Migrieren in einer SQL Server-Datenbank in zwei miteinander per 1:n-Beziehung verknüpfte Tabellen umgewandelt werden. Die 1:1-Beziehung ist ein Spezialfall der 1:n-Beziehung, für die eine kleine Sonderbehandlung notwendig ist. Wie des gelingt, zeigt der vorliegende Beitrag.

Vorbereitungen

Für diesen Artikel nutzen wir ein Projekt auf Basis der Vorlage **Visual C#|Windows Desktop|WPF-App** namens **EDMConfigure11**. Die Techniken lassen sich aber auch in Projekten auf Basis anderer Vorlagen nutzen, beispielsweise in Webanwendungen. Wir fügen wie im Artikel EDM: 1:n-Beziehungen per Code First ein ADO.NET Entity Data Model hinzu, wieder namens **CustomerManagementContext**, und starten auch hier wieder mit einem leeren Code First-Modell. Das Attribut **initial catalog** in der Verbindungszeichenfolge in der Datei **App.conf** stellen wir auf **EDMConfigure11.CustomerManagement** ein.

Beispiel

Unser einfaches Beispiel soll eine Klasse namens **Customer** und eine namens **CustomerInvoiceAddress** enthalten und daraus die Tabellen **Customers** und **CustomerInvoiceAddresses** erzeugen, die über die Felder **ID** der Tabelle **Customers** und das gleichnamige Feld der Tabelle **CustomerInvoiceAddresses** miteinander verknüpft sind.

Klassen erstellen und DbContext einrichten

In einem neuen Verzeichnis namens **Data** legen wir die als öffentlich deklarierte Klasse namens **Customer.cs** mit dem folgenden Code an:

```
namespace CustomerManagement.Data {
    public class Customer {
        public int ID { get; set; }
        public string Company { get; set; }
        public string FirstName { get; set; }
        public string LastName { get; set; }
    }
}
```

Auf die gleiche Weise erzeugen wir eine Klasse für die Rechnungsadressen, die wir **CustomerInvoiceAddress** nennen. Auch diese legen wir im Verzeichnis **Data** an und stellen die Sichtbarkeit auf öffentlich ein:

Nun müssen wir noch die **DbSet**-Elemente in der Klasse **CustomerManagementContext** hinzufügen. Damit wir dabei auf die Klassen **Customer** und **Salutation** als Datentypen der Auflistungen zugreifen können, machen wir den Namespace **CustomerManagement.Data** mit der **using**-Anweisung verfügbar:

```
namespace EDMConfigure11.Data {
    public class CustomerInvoiceAddress {
        public int ID { get; set; }
        public string Street { get; set; }
        public string Zip { get; set; }
        public string City { get; set; }
    }
}
```

Die Datei **CustomerManagementContext.cs** passen wir nun wie folgt an. Dabei fügen wir per **using**-Anweisung den Namespace **EDMConfigure11.Data** an, damit wir auf die in dem Verzeichnis **Data** enthaltenen Klassen, die dort auch mit dem entsprechenden Namespace versehen sind, zugreifen können. Außerdem erweitern wir die Klasse um die beiden **DbSet**-Elemente namens **Customers** und **CustomersInvoiceAddresses**:

```
namespace EDMConfigure11 {
    ...
    using EDMConfigure11.Data;
    public class CustomerManagementContext : DbContext {
        public CustomerManagementContext()
            : base("name=CustomerManagementContext") {
        }
        public virtual DbSet<Customer> Customers { get; set; }
        public virtual DbSet<CustomerInvoiceAddress> CustomerInvoiceAddresses { get; set; }
    }
}
```

Datenbank erstellen

Nun migrieren wir die Klassen erstmals in die Zieldatenbank, die noch zu erstellen ist. Um das Migrieren aus dem Projekt heraus zu ermöglichen, rufen wir zunächst die folgende Anweisung in der Paket-Manager-Konsole auf, was das Verzeichnis **Migrations** mit der Datei **Configure.cs** erstellt:

```
enable-migrations
```

Dann erstellen wir das Skript für die erste Migration, was eine Datei zum Verzeichnis **Migrations** hinzufügt, die auf **_Init.cs** endet:

```
add-migration Init
```

Schließlich erstellen wir erstmalig die Datenbank:

```
update-database
```

Dies erstellt die beiden Tabellen **Customers** und **CustomerInvoiceAddresses** in der neuen Datenbank **EDMConfigure11.CustomerManagement**. Allerdings enthalten diese noch keinerlei Beziehung zueinander, wie Bild 1 zeigt.

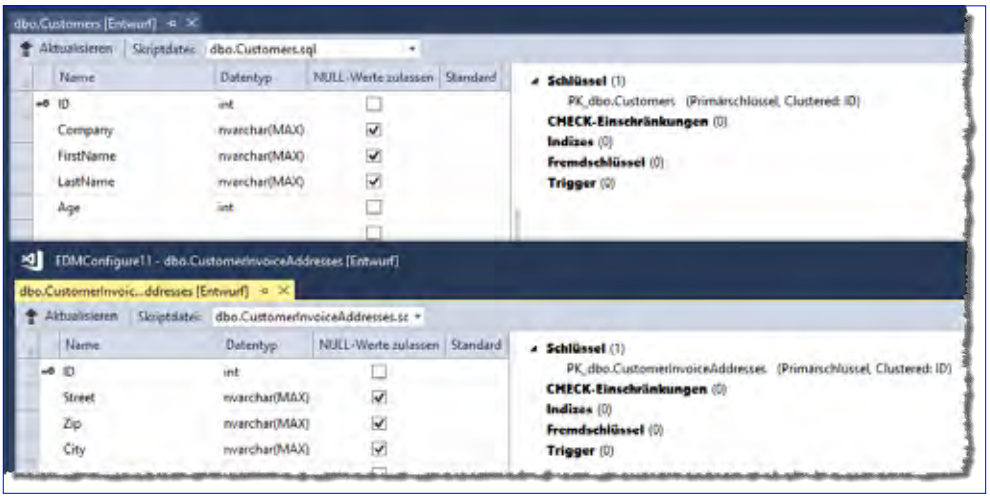


Bild 1: Definitionen der Tabellen **Customers** und **CustomerInvoiceAddresses**

1:1-Beziehung erzeugen

Wenn wir bei der 1:n-Beziehung der Klasse, welche die Seite der Beziehung mit dem Fremdschlüsselfeld der Beziehung bilden soll, eine Eigenschaft des Typs der zu verknüpfenden Klasse zugewiesen haben, kann dies auch hier nicht verkehrt sein. Also fügen wir die Eigenschaft **CustomerInvoiceAddress** mit dem entsprechenden Namen zu **Customer.cs** hinzu:

```
public class Customer {
    public int ID { get; set; }
    ...
    public CustomerInvoiceAddress CustomerInvoiceAddress { get; set; }
}
```

Bei der 1:n-Beziehung haben wir der Klasse für die Seite der Beziehung mit dem Primärschlüsselfeld, hier **Salutation**, eine **ICollection** mit den Elementen des Typs **Customer** hinzugefügt. Der Logik folgend sollten wir bei der 1:1-Beziehung dann keine **ICollection**, sondern ebenfalls eine Eigenschaft mit dem Typ des zu referenzierenden Elements hinzufügen. Das sieht dann für die Klasse **CustomerInvoiceAddress** wie folgt aus:

```
public class CustomerInvoiceAddress {
    public int ID { get; set; }
    ...
    public Customer Customer { get; set; }
}
```

Damit wollen wir nun eine erneute Migration wagen und geben die Anweisung **add-migration AddRelation** in die Paket-Manager-Konsole ein. Als Ergebnis liefert diese die folgende Fehlermeldung:

```
Das Prinzipalende einer Zuordnung zwischen den Typen 'EDMConfigure11.Data.Customer' und 'EDMConfigure11.Data.CustomerInvoiceAddress' konnte nicht ermittelt werden. Das Prinzipalende dieser Zuordnung muss mithilfe der Fluent-API für Beziehungen oder mithilfe von Datenanmerkungen explizit konfiguriert werden.
```

EF Core: Klassendiagramm anzeigen

Für die kleinen Beispiele, die wir bisher programmiert haben, benötigen Sie keine grafische Übersicht. Wenn das Projekt aber wächst und sich immer mehr Klassen ansammeln, macht es Sinn, die Klassen und ihre Beziehungen untereinander in einem Diagramm anzuzeigen. Dieser Artikel zeigt, wie Sie das mit den Bordmitteln von Visual Studio ganz einfach erledigen.

Wenn Sie ein Entity Data Model auf Basis der Vorlage **EF Designer aus Datenbank** erstellen, haben Sie keine Sorgen – das Klassendiagramm wird dann automatisch bereitgestellt. Es handelt sich um die **.edmx**-Datei, die Sie einfach zu öffnen brauchen.

Wenn Sie hingegen, wie wir es in den Entity Framework-Artikeln der laufenden Ausgaben tun, die Vorlage **Leeres Code First-Modell** nutzen, um das Modell zunächst manuell zu erstellen und erst dann die Datenbank daraus erzeugen zu lassen, finden Sie diese Datei nicht automatisch vor. Es gibt allerdings die Möglichkeit, diese Datei erzeugen zu lassen – der Einfachheit halber wollen wir das direkt beim Starten der Anwendung erledigen.

Voraussetzungen

Voraussetzung für die nachfolgend beschriebene Vorgehensweise ist, dass Ihr Projekt mindestens eine Klasse enthält, die von der Klasse **DbContext** abgeleitet ist. Eine solche Klasse wird automatisch erstellt, wenn Sie dem Projekt ein Entity Data Modell etwa auf Basis der Vorlage **Leeres Code First-Modell** hinzufügen – etwa so:

```
public class Customer {
    public int ID { get; set; }
    public string Company { get; set; }
    public string FirstName { get; set; }
    public string LastName { get; set; }
}
```

Sie benötigen ein oder mehrere Entitätsklassen, die Sie durch entsprechende Anweisungen in der **DbContext**-Klasse in **DbSet**-Auflistungen erfassen, beispielsweise wie folgt:

```
public class CustomerManagementContext : DbContext {
    public CustomerManagementContext() : base("name=CustomerManagementContext") {
    }
    public virtual DbSet<Customer> Customers { get; set; }
    }
}
```

Das unten gezeigte Klassendiagramm enthält noch weitere Objekte und Beziehungen, die wir hier aus Gründen der Übersicht nicht als Code abbilden wollen.

Schnellstart mit Datenbank

In einigen Artikeln haben wir ausführlich und Schritt für Schritt erklärt, wie Sie eine Webanwendung mit Datenbankanbindung anlegen: Sie erstellen die Klassen für die einzelnen Entitäten, fügen eine Datenbankkontextklasse hinzu, legen ein paar andere notwendige Elemente an und erstellen dann mit dem Gerüstbau-Assistenten die gewünschten Seiten an, die Sie dann nach eigenen Vorstellungen erweitern können. Einige dieser Schritte können Sie sich, wenn Sie die Funktion verstanden haben, auch sparen – nach dem Anlegen der Klassen können Sie nämlich einige Schritte auf einen Schlag erledigen. Wie das gelingt, zeigt der vorliegende Beitrag.

Für die Beispiele dieses Artikels haben wir ein neues Projekt auf Basis der Vorlage Visual **C#|Web|ASP.NET Core-Webanwendung** für **ASP.NET Core 2.1** mit der Projektvorlage **Webanwendung** angelegt – ohne Hinzufügen einer Authentifizierung. Einem neuen Verzeichnis namens **Models** haben wir eine neue Klasse namens **Article.cs** hinzugefügt, welche die Artikel-Entitäten aufnehmen soll:

```
public class Article {
    public int ID { get; set; }
    public string Name { get; set; }
    public decimal Price { get; set; }
    public int CategoryID { get; set; }
    public Category Category { get; set; }
}
```

Eine weitere Klasse namens **Category** übernimmt die Kategorien:

```
public class Category {
    public int ID { get; set; }
    public string Name { get; set; }
    public virtual ICollection<Article> Article { get; set; }
}
```

Die beiden Klassen sind über die Navigationseigenschaft **Category** beziehungsweise die **ICollection**-Auflistung **Article** miteinander verknüpft. Dann führen wir ein paar Schritte, die wir in anderen Artikeln manuell durchgeführt haben, direkt über den Gerüst-Generator aus. Dazu legen Sie ein neues Verzeichnis namens **Articles** unterhalb des Verzeichnisses **Pages** an. Wählen Sie aus dem Kontextmenü des neuen Eintrags den Befehl **Hinzufügen|Neues Gerüstelement...** aus. Im nun erscheinenden Dialog **Gerüst hinzufügen** aktivieren Sie die Option **Razor-Seiten mithilfe des Entity Frameworks (CRUD)** und klicken auf **Hinzufügen**. Da wir noch nicht, wie in anderen Beispielen, manuell eine Datenkontextklasse erstellt haben, klicken wir im Dialog aus Bild 1 auf die Schaltfläche mit dem **Plus**-Symbol neben der Eigenschaft **Datenkontextklasse**. Im nun erscheinenden Input-Fenster geben wir den Namen ein – beispielsweise wie hier vorgeschlagen **Schnellstart.Models.SchnellstartContext**. Diesen neuen Eintrag müssen

Sie dann noch für die Eigenschaft Datenkontextklasse auswählen und den Erstellungsvorgang mit einem Klick auf die Schaltfläche **Hinzufügen** starten. Danach arbeitet Visual Studio für einige Sekunden an der Erstellung der notwendigen sowie an der Anpassung vorhandener Dateien.

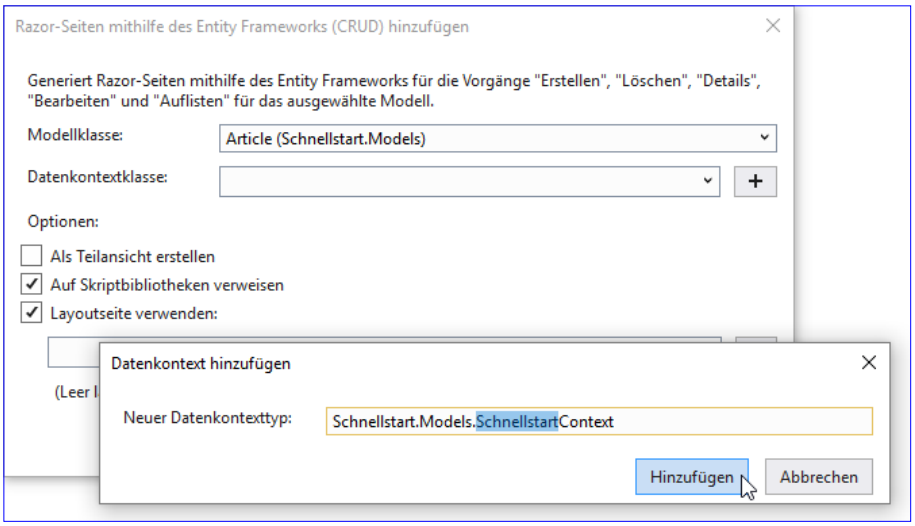


Bild 1: Anlegen einer neuen Datenkontextklasse

Was ist geschehen?

Bevor wir die Anwendung einfach einmal starten, schauen wir uns die Veränderungen in den Projektdateien an. Als Erstes fällt uns natürlich auf, dass im Verzeichnis **Pages/Articles** einige neue Dateien wie **Create.cshtml** und so weiter erscheinen – aber die waren uns ja schon von früheren Artikeln bekannt. Vorher haben wir aber noch einige Änderungen manuell durchgeführt. Diese wurden nun automatisch hinzugefügt, sofern noch nicht vorhanden. Schauen wir uns an, was noch geschehen ist. Da wäre zunächst die von uns gewünscht Datenkontextklasse. Diese wurde in einem neuen Verzeichnis namens **Data** angelegt, wurde aber dem Namespace **Schnellstart.Models** zugeordnet. Die Klasse enthält die noch leere Konstruktormethode **SchnellstartContext** sowie das **DbSet** namens **Article** (siehe Bild 2). Die Frage, die sich hier stellt, ist: Warum finden wir hier nur ein **DbSet** auf Basis der Klasse **Article** vor, aber keine auf Basis der Klasse **Category**? Der Grund ist, dass die Klasse **Article** mit der Navigationseigenschaft **Category** auf die Klasse **Category** verweist. Der Ordnung halber können Sie die Klasse **Category** aber dennoch aufnehmen. Spätestens, wenn Sie eine Klasse hinzufügen, die nicht mit **Article** oder einer anderen bereits als **DbSet** registrierten Klasse verknüpft ist, müssen Sie dafür ein eigenes **DbSet** anlegen. Außerdem wollen wir den Namen des **DbSet** noch auf **Articles** ändern. Leider müssen Sie danach auch noch die Stellen, an denen die Lösung auf das DbSet **Article** zugreift, in **Articles** umbenennen.

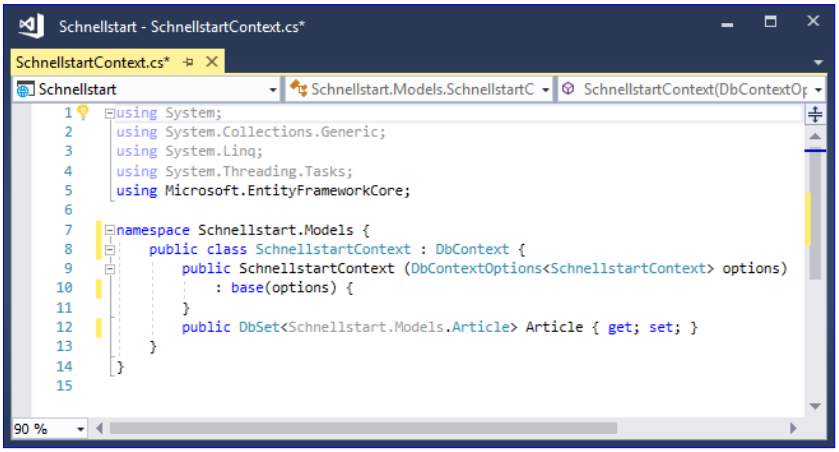


Bild 2: Die neue Datenkontextklasse

Die Datei **appsettings.json** mit den Konfigurationsdaten wurde wie folgt um ein **ConnectionString**-Element erweitert – hier haben wir den Namen der Datenbank noch etwas angepasst:

```
{
  "Logging": {
```

Authentifizierung nachrüsten

Im Artikel »Authentifizierung unter ASP.NET Core« haben wir gezeigt, wie Sie eine ASP.NET Core-Anwendung erstellen und diese direkt mit Funktionen für die Authentifizierung ausstatten. In anderen Artikeln wiederum haben wir Lösungen beschrieben, in denen wir zunächst ohne die Authentifizierung gestartet sind. Nun wollen wir beides zusammenführen. Aber was tun, wenn die Lösung noch nicht die für die Authentifizierung notwendigen Elemente enthält – nachrüsten oder neu mit Authentifizierung erstellen und dann die Lösung übertragen? Dieser Artikel zeigt den einfacheren Weg.

Voraussetzungen: .NET Core 2.1

Der Assistent zum Erstellen neuer ASP.NET Core-Anwendung liefert eine wirklich brauchbare Basis für neue Anwendungen. Noch schöner ist, dass er das Framework für die Realisierung einer Authentifizierung optional mitliefert und gleich noch die Links zur Registrierung und zur Anmeldung sowie die zur Verwaltung notwendigen hinzufügt.

Wenn Sie allerdings mit der Programmierung einer Lösung begonnen haben und nicht gleich die Option zum Hinzufügen der Authentifizierung aktiviert haben, stehen Sie erst einmal ohne Authentifizierung da. Und dazu gehören eine ganze Menge Elemente: die Datenbank zum Speichern der Benutzerdaten, die Verweise, einige Dateien mit den Seiten für Login, Registrierung und so weiter und letztlich auch noch die Einbindung der Links zum Einloggen und Registrieren in die Navigationsleiste der Startseite. Können wir diese Elemente so einfach nachrüsten und in eine bestehende ASP.NET Core-Anwendung integrieren? Oder macht es mehr Sinn, die Anwendung mit Authentifizierung neu zu erstellen und die bestehenden Elemente zu übertragen? Wenn Sie erst ein paar Seiten erstellt haben, können Sie sicher den letzteren Weg einschlagen. Je größer die Lösung allerdings schon gewachsen ist, desto geringer wird der Aufwand für das nachträgliche Einbauen der Authentifizierung gegenüber der anderen Alternative sein. Also schauen wir uns diesen Ansatz einmal genauer an.

Beispielprojekt erstellen

Als Erstes legen wir ein Beispielprojekt an. Dazu nutzen wir den Dialog **Neues Projekt**, wo wir den Eintrag **Visual C#|Web|ASP.NET Core-Webanwendung** auswählen (siehe Bild 1). Außerdem wollen wir unsere Lösung **AuthNachruesten** nennen.

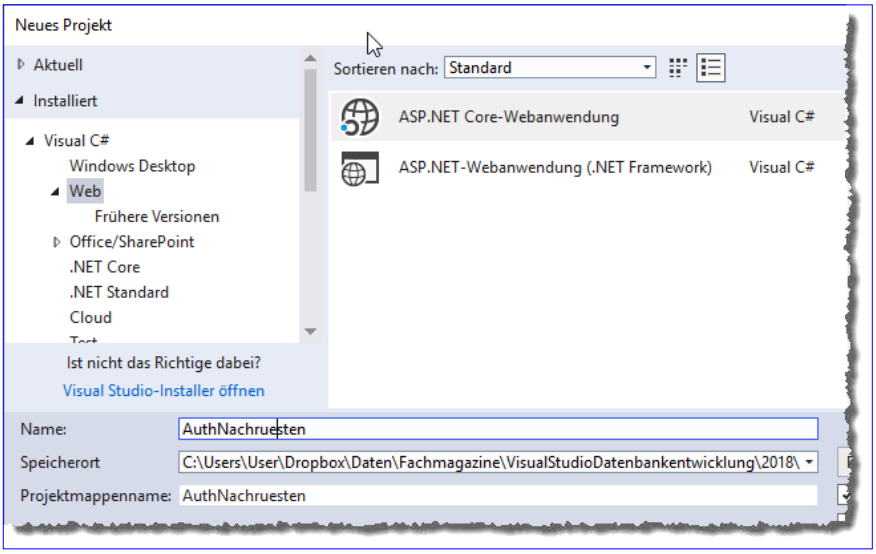


Bild 1: Projekt erstellen

Im nächsten Dialog behalten wir die Auswahl der Option **Webanwendung** bei. Hier wäre die Möglichkeit, über die Schaltfläche **Authentifizierung** ändern

direkt die Authentifizierung hinzuzufügen. Dies wollen wir aber auslassen, da das Ziel des Artikels ja gerade die manuelle Nachrüstung der Authentifizierung ist (siehe Bild 2).

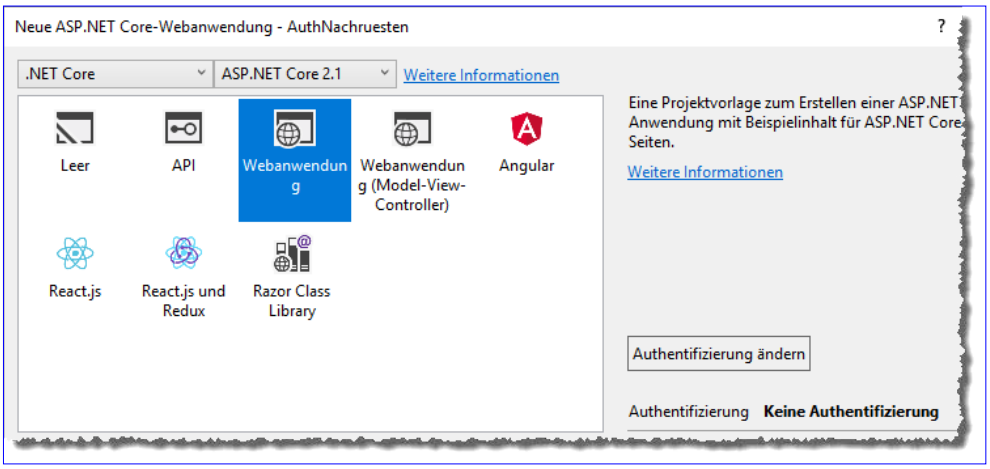


Bild 2: Auswahl ohne Authentifizierung

Wenn Sie die Anwendung nun starten, sehen Sie nur die Standardnavigation (siehe Bild 3). Wir hätten gern, dass oben rechts Navigationseinträge angezeigt werden, mit denen sich Benutzer registrieren und einloggen können. Dies rüsten wir nun nach.

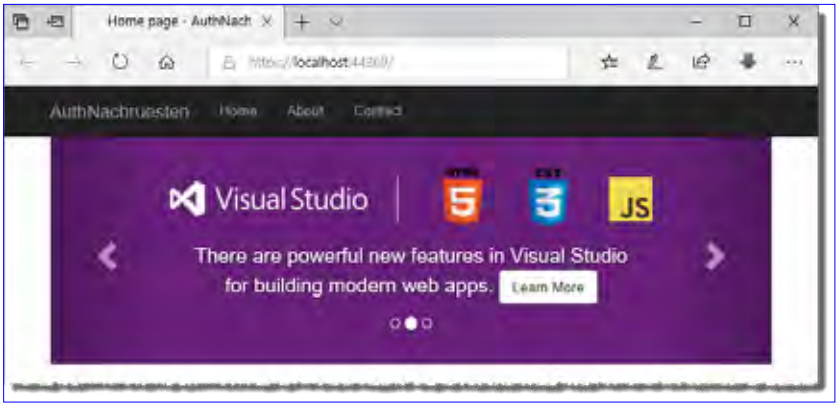


Bild 3: Start der Webanwendung ohne Authentifizierung

Notwendiges Paket nachinstallieren

Das Paket, dass die notwendigen Funktionen enthält, heißt **Microsoft.AspNetCore.Identity.EntityFrameworkCore** – es handelt sich dabei um ein NuGet-Paket. Diese können Sie leicht nachrüsten – beispielsweise über den dafür vorgesehenen NuGet-Paket-Manager.

Diesen starten Sie über den Kontextmenü-Eintrag **NuGet-Pakete verwalten...**. Es erscheint der Dialog aus Bild 4, wo Sie zum Bereich **Durchsuchen** wechseln und dort den Suchbegriff **Microsoft.AspNetCore.Identity.EntityFrameworkCore** eingeben. Haben Sie den entsprechenden Eintrag gefunden, installieren Sie diesen mit einem Klick auf die Schaltfläche **Installieren**.

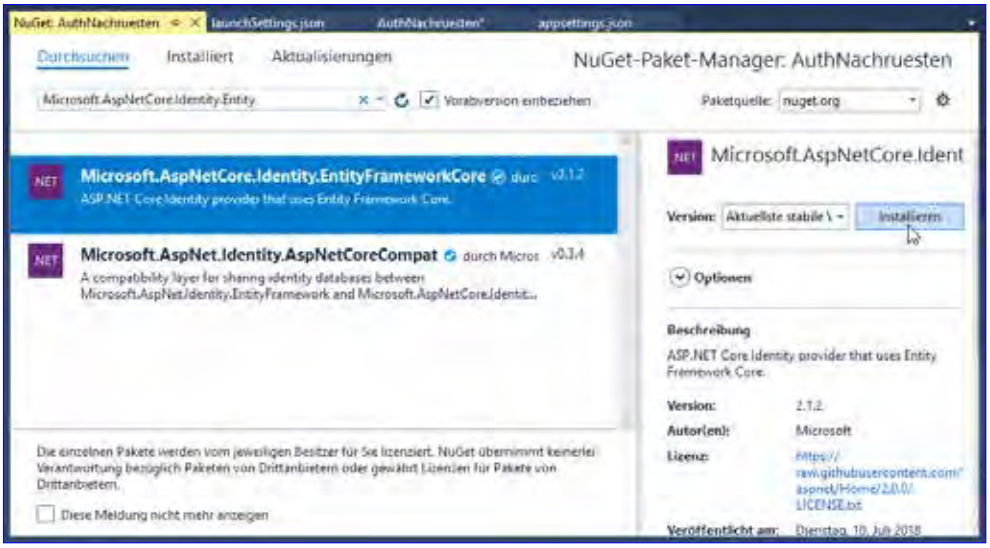


Bild 4: **Microsoft.AspNetCore.Identity.EntityFrameworkCore** nachrüsten

Microsoft holt dann noch Ihre Zustimmung zur Lizenz ein, die Sie mit einem Klick auf die Schaltfläche **Ich stimme zu** quittieren. Die entsprechende Meldung zeigt gleich noch an, welche weiteren Bibliotheken mit dieser Bibliothek installiert werden (siehe Bild 5).

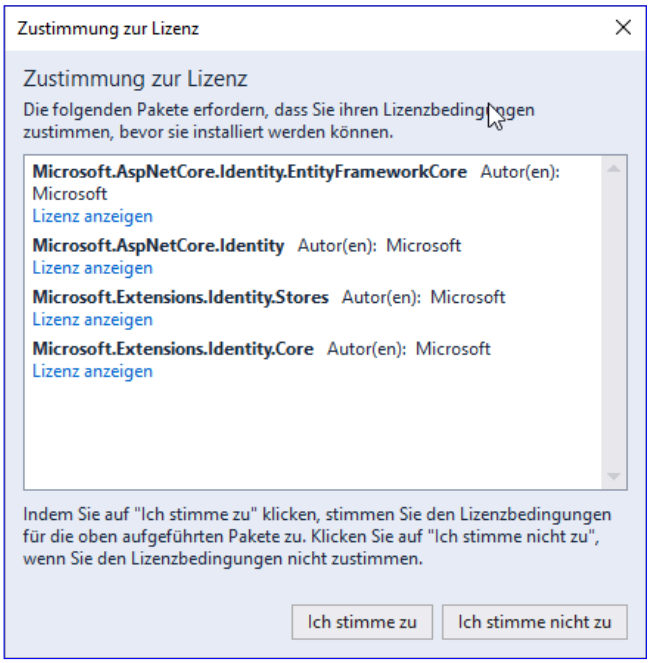


Bild 5: Microsoft.AspNetCore.Identity.EntityFrameworkCore kommt mit ein paar weiteren Bibliotheken

Elemente für CodeFirst-Datenbank hinzufügen

Wenn wir mit der Authentifizierung arbeiten wollen, wir uns die Bibliotheken zur Verfügung stellen, haben wir mehrere Möglichkeiten, was die Speicherung der damit in Zusammenhang stehenden Daten angeht. Wir wollen es klassisch machen und die Daten in einer SQL Server-Datenbank speichern. Dabei gehen wir nach der Code First-Methode vor und erstellen erst ein paar Elemente im Projekt. Danach erstellen wir auf Basis dieser Objekte die notwendige Datenbank beziehungsweise lassen diese erstellen.

Genaugenommen bauen wir ein Entity Data Model auf, dessen Elemente wir in einem Unterverzeichnis namens **Data** im Projektordner anlegen. Das Verzeichnis **Data** legen Sie an, indem Sie den Kontextmenü-Eintrag **Hinzufügen|Neuer Ordner** des Projekt-Eintrags im Projektmappen-Explorer aufrufen.

Dem Ordner **Data** fügen Sie über den Kontextmenü-Eintrag **Hinzufügen|Klasse** eine neue Klasse hinzu, die Sie **ApplicationUser** nennen. Diese Klasse füllen Sie wie folgt:

```
01.     using System;
02.     using System.Collections.Generic;
03.     using System.Linq;
04.     using System.Threading.Tasks;
05.     using Microsoft.AspNetCore.Identity;
06.
07.     namespace AuthNachruesten.Data {
08.         public class ApplicationUser : IdentityUser {
09.         }
10.     }
```

Wir fügen also den Namespace **Microsoft.AspNetCore.Identity** hinzu und stellen die Klasse **Application** so ein, dass Sie von **IdentityUser** abgeleitet wird.

Auf die gleiche Weise fügen Sie eine weitere Klasse namens **ApplicationDbContext** hinzu, welche den Datenbank-Kontext für das Entity Data Model enthalten soll. Diese passen Sie wie folgt an:

```
using System;
using System.Collections.Generic;
using System.Linq;
using System.Threading.Tasks;
using Microsoft.AspNetCore.Identity.EntityFrameworkCore;
using Microsoft.EntityFrameworkCore;

namespace AuthNachruesten.Data {
    public class ApplicationDbContext : IdentityDbContext<ApplicationUser> {
        public ApplicationDbContext(DbContextOptions<ApplicationDbContext> options): base(options) {
        }
        protected override void OnModelCreating(ModelBuilder builder) {
            base.OnModelCreating(builder);
        }
    }
}
```

Nun nehmen wir ein paar Änderungen an der Klasse **Startup.cs** vor. Wir benötigen einen Verweis auf den Namespace **Microsoft.AspNetCore.Identity**, einen auf **Microsoft.EntityFrameworkCore** sowie einen auf unseren Namespace mit dem Entity Data Model, also **AuthNachruesten.Data**. Schließlich fügen wir vor der Anweisung **services.AddMvc()...** die Methode **services.AddDbContext...** sowie die Methode **services.AddIdentity...** ein:

```
...
using Microsoft.AspNetCore.Identity;
using Microsoft.EntityFrameworkCore;
using AuthNachruesten.Data;

namespace AuthNachruesten {
    public class Startup {
        ...
        public void ConfigureServices(IServiceCollection services) {
            ...
            services.AddDbContext<ApplicationDbContext>(
                options => options.UseSqlServer(Configuration.GetConnectionString("DefaultConnection")));
            services.AddIdentity<ApplicationUser, IdentityRole>()
                .AddEntityFrameworkStores<ApplicationDbContext>()
                .AddDefaultTokenProviders();
            services.AddMvc().SetCompatibilityVersion(CompatibilityVersion.Version_2_1);
        }
    }
}
```

An dieser Stelle verwenden wir eine Verbindungszeichenfolge namens **DefaultConnection**. Diese ist allerdings noch nicht hinterlegt, was wir wie folgt in der Datei **appsettings.json** erledigen:

```
{
  "ConnectionStrings": {
    "DefaultConnection": "Server=(localdb)\\mssqllocaldb;Database=AuthNachruesten;Trusted_Connection=True;
                         MultipleActiveResultSets=true"
  },
  "Logging": {
    "LogLevel": {
      "Default": "Warning"
    }
  },
  "AllowedHosts": "*"
}
```

Datenbank erstellen

Um eine Datenbank auf Basis der beiden Klassen **ApplicationDbContext.cs** und **ApplicationUser.cs** zu erstellen, benötigen wir die Paket-Manager-Konsole. Diese blenden wir über den Menüeintrag **Extras|NuGet-Paket-Manager|Paket-Manager-Konsole** ein. Nun geben wir in der Paket-Manager-Konsole die folgende Anweisung ein (siehe Bild 6):

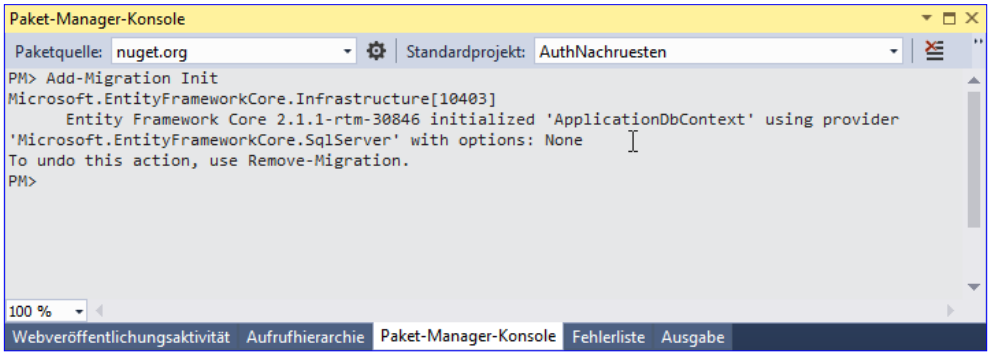


Bild 6: Initialisieren der Migration

```
Add-Migration Init
```

Danach folgt dieser Aufruf:

```
Update-Database
```

Visual Studio arbeitet nun ein paar Sekunden und meldet dann den erfolgreichen Abschluss der Aufgabe. Wenn wir nun den SQL Server-Objekt-Explorer einblenden, finden wir dort unter dem angegebenen SQL Server, hier **LocalDb**, die neue Datenbank **AuthNachruesten** vor. Klappen wir diesen Eintrag und den darin enthaltenen Eintrag **Tabellen** auf, finden wir eine Reihe Tabellen wie in Bild 8 vor. Interessanterweise ist hier keine Tabelle namens **ApplicationUser** zu sehen. Aber den haben wir in der Datei **ApplicationDbContext** ja auch gar nicht als **DbSet** definiert. Aber woher wusste Visual Studio, welche Tabellen hier erstellt werden sollten? Es handelt sich ja offensichtlich um die für die Verwaltung von Benutzern notwendigen Tabellen, ganz verkehrt liegen wir hier also nicht.

Authentifizierungsseiten anpassen

Im Artikel »Authentifizierung nachrüsten« haben wir gezeigt, wie Sie die Funktionen für die Authentifizierung von Benutzern in einer Webanwendung nachrüsten können. Dort haben wir die platzsparende Variante verwendet – also die, welche die ganzen verwendeten Elemente der Benutzeroberfläche für die Authentifizierung direkt aus einer Bibliothek verwendet. Das spart zwar eine Menge Platz, weil die ganzen Seiten nicht einzeln im Projektordner liegen, aber man kann diese nicht so einfach anpassen. Der vorliegende Artikel zeigt, wie Sie die Authentifizierung umstellen und diese so anpassbar machen.

Im Artikel **Authentifizierung nachrüsten** haben wir die unter ASP.NET Core 2.1 eingeführte Bibliothek **Microsoft.AspNetCore.Identity.UI** eingebunden, um die darin enthaltenen Elemente für die Benutzer-Authentifizierung zu nutzen. Damit haben wir den Umfang der im Projektordner enthaltenen Daten erheblich verringert, denn die Seiten für die Anzeige der Elemente der Authentifizierung und erst recht der Code für die notwendigen Funktionen sind sehr umfangreich (rund 50 Dateien). Diese Bibliothek haben wir verfügbar gemacht, in dem wir in der Klasse **Startup.cs** die Methode **AddDefaultUI()** aufrufen.

Nun wollen wir allerdings beispielsweise die Texte in deutscher Sprache präsentieren und nicht in englischer Sprache. Dazu müssen wir allerdings doch wieder den Code für die Authentifizierungsfunktionen ins Projekt holen. Hier gibt es zwei verschiedene Ansätze: Entweder Sie holen den kompletten Code an Bord oder Sie fügen nur diejenigen Dateien ein, die Sie anpassen möchten, um beispielsweise die Benutzeroberfläche in deutscher Sprache zu präsentieren. Wir zeigen Ihnen in den nächsten Abschnitten, wie das funktioniert.

Authentifizierungsfunktionen ins Projekt holen

Um die Elemente der Bibliothek **Microsoft.AspNetCore.Identity.UI** doch wieder im Projektverzeichnis unterzubringen, benötigen Sie ein neues Element. Diesmal fügen wir allerdings nicht einfach ein Element etwa wie eine Klasse hinzu, sondern verwenden einen sogenannten Scaffolded Item, zu deutsch Gerüstelement. Dazu rufen Sie zunächst den Kontextmenü-Befehl **Hinzufügen|Neues Gerüstelement...** auf. Es erscheint der Dialog **Gerüst hinzufügen**, in dem wir links **Identität** auswählen und in der Mitte anschließend auch (siehe Bild 1).

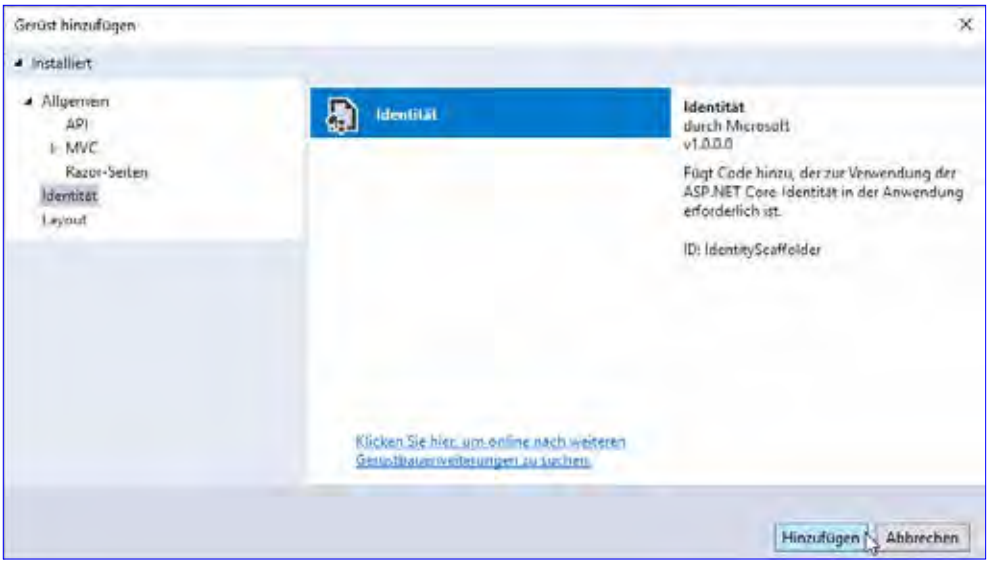


Bild 1: Hinzufügen der Authentifizierungs-Elemente per Gerüstbau

Danach erscheint der Dialog aus Bild 2. Hier haben Sie verschiedene Möglichkeiten:

- Angabe der Layout-Seite
- Auswahl der zu überschreibenden Dateien – gegebenenfalls auch die Möglichkeit, alle Dateien zu überschreiben
- Auswahl der Kontext-Klasse
- Angabe der Benutzerklasse

Die Angabe der Layout-Seite können wir weglassen, da wir ja bereits in der Datei **Areas\Identity\Pages** eine Datei namens **_ViewStart.cshtml** angelegt haben, die einen Verweis auf die zu verwendende Layout-Datei enthält (**Pages\Shared_Layout.cshtml**).

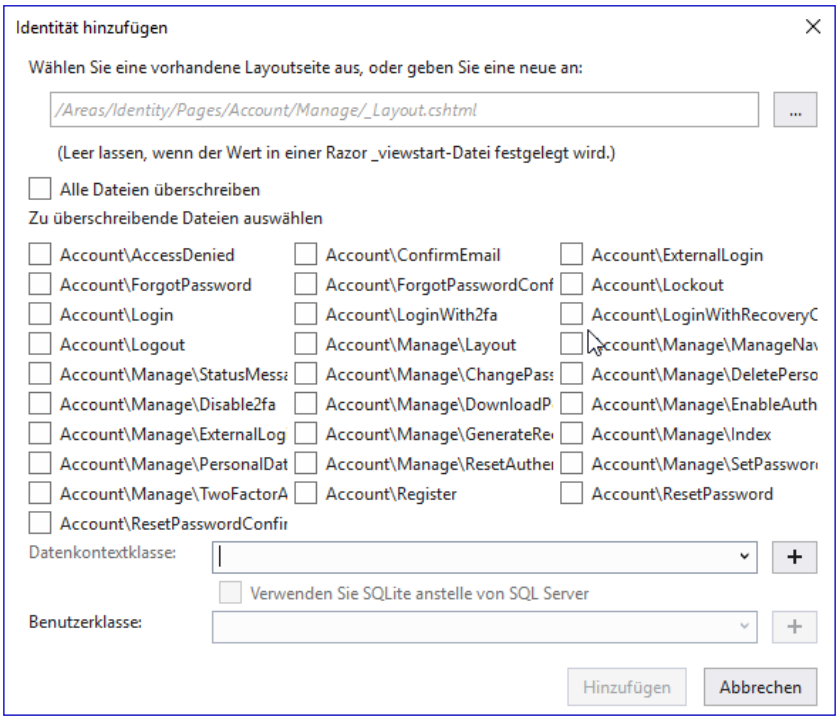


Bild 2: Konfiguration zum Hinzufügen der Elemente

Darunter könne Sie auswählen, welche der vielen Dateien, die sich in der Bibliothek **Microsoft.AspNetCore.Identity.UI** befinden, Sie überschreiben wollen. Überschreiben deshalb, weil ja die Dateien in der Bibliothek noch vorhanden sind, aber stattdessen dann die von Ihnen hinzugefügten Dateien verwendet werden. Wir wollen zunächst die offensichtlichen Dateien hinzufügen:

- **Account\Login**
- **Account\Register**

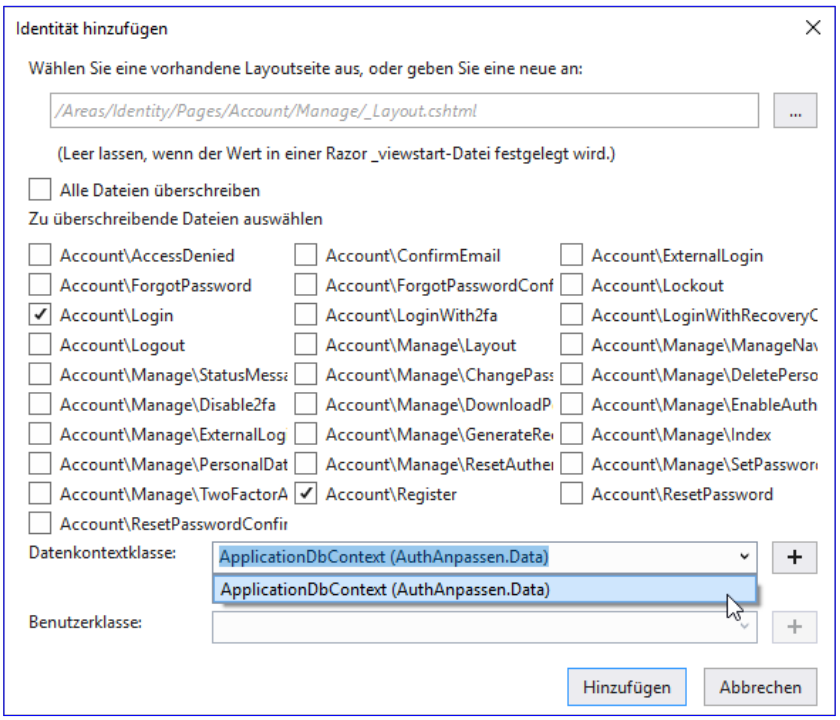


Bild 3: Auswahl der hinzuzufügenden Elemente

Erst wenn Sie hier mindestens eine Datei ausgewählt haben, wird das Auswahlfeld für die Datenkontextklasse gefüllt. Hier wählen wir dann unsere Datenkontextklasse **ApplicationDbContext** aus dem Verzeichnis **AuthAnpassen.Data** aus (siehe Bild 3). Das Textfeld Benutzerklasse wird nicht aktiviert und kann ignoriert werden.

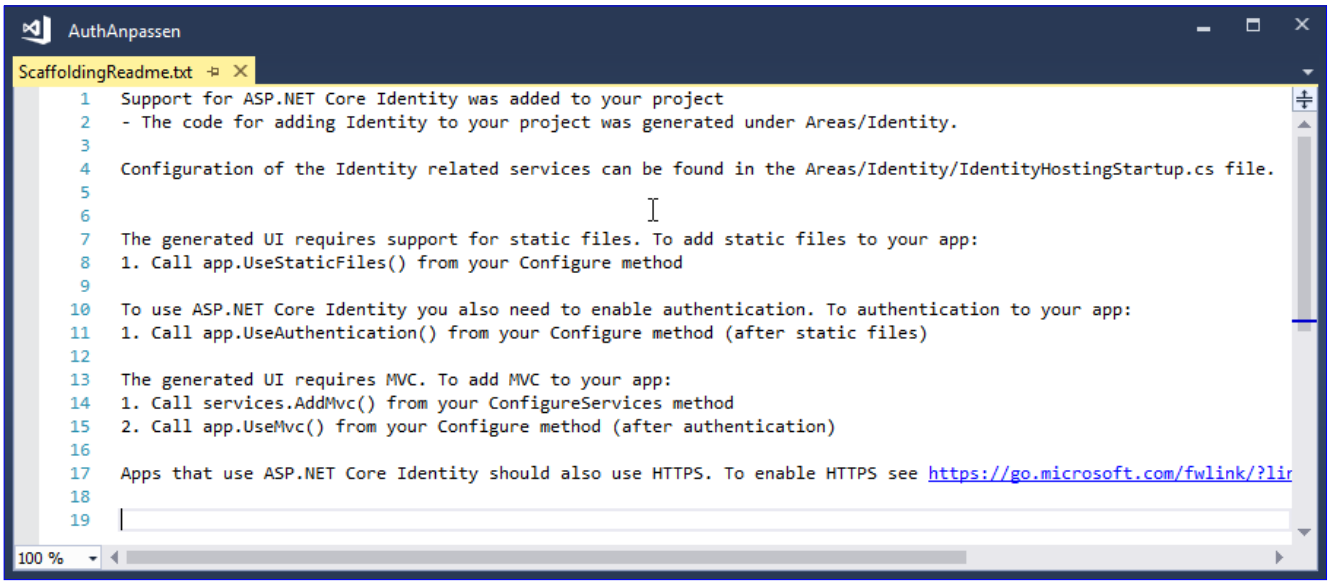


Bild 4: Hinweise nach dem Gerüstbau

Klicken Sie dann auf **OK**, erscheint ein Dialog, der den Fortschritt dokumentiert. Außerdem zeigt Visual Studio nach Abschluss des Vorgangs noch eine Readme-Datei an (siehe Bild 4). Demnach soll die **Configure**-Methode der Klasse **Startup.cs** die Methode **app.UseStaticFiles()** aufrufen. Dahinter soll die Methode **app.UseAuthentication()** aufgerufen werden sowie danach die Methode **app.UseMvc**.

Die ebenfalls in diese Klasse enthaltene Methode **ConfigureServices** soll **services.AddMvc**() aufrufen. Prüfen Sie, ob die angegebenen Methoden vorhanden sind (was der Fall sein sollte) und ergänzen Sie diese gegebenenfalls.

Dateien anpassen

Der Gerüstbauer hat außerdem ganze Arbeit geleistet, wie Bild 5 zeigt. Er hat unterhalb von **Areas\Identity\Pages** das Verzeichnis **Account** mit den gewünschten Dateien, hier **Login.cshtml** und **Register.cshtml**, und einigen weiteren Dateien angelegt. Außerdem einen Ordner namens **Services** unterhalb von **Areas\Identity** sowie die Datei **IdentityHostingStartup.cs**.

Wenn wir nun etwa die Texte der Seite **Login** eindeutschen wollen, brauchen wir nur die Datei **Login.cshtml** zu öffnen und dort die gewünschten Änderungen vorzunehmen.

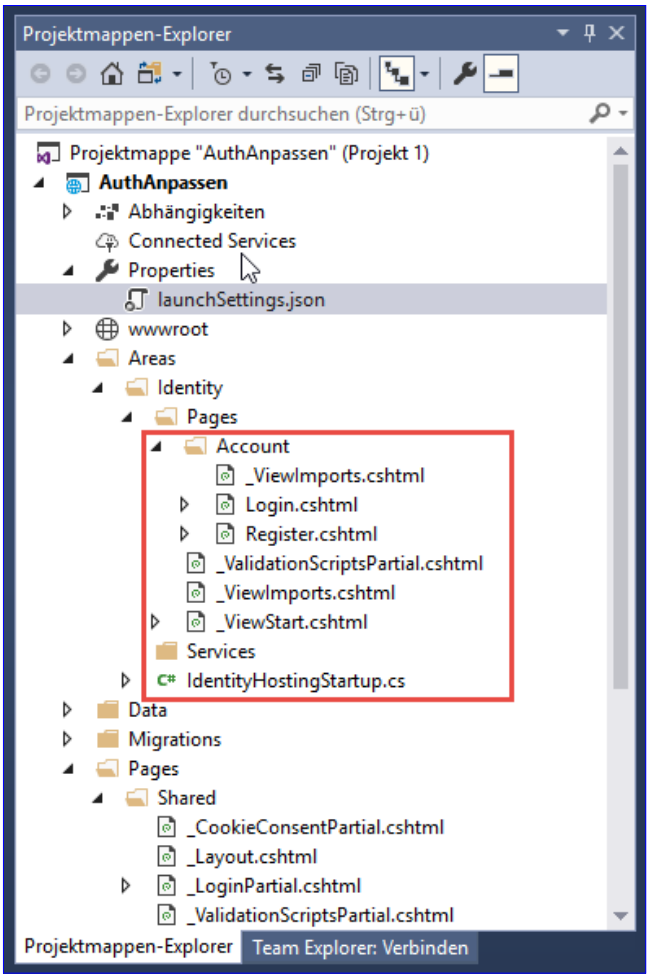


Bild 5: Ergebnis des Gerüstbaus

Authentifizierung um Felder erweitern

Die Authentifizierung spielt eine wichtige Rolle bei der Programmierung von Webanwendungen, deren Inhalte nicht ausschließlich der Allgemeinheit zur Verfügung stehen sollen. Dabei kann es vorkommen, dass es nicht ausreicht, die E-Mail-Adresse und das Kennwort eines Benutzers zu speichern. Manchmal möchte man es vielleicht persönlicher gestalten und den Benutzer direkt ansprechen. Um bei der Registrierung die notwendigen Daten zu speichern, sind auch bei der ansonsten perfekt funktionierenden Registrierung von ASP NET Core noch einige Erweiterungen durchzuführen.

Voraussetzungen: Ausgangspunkt für die in diesem Artikel beschriebenen Erweiterungen ist eine Webanwendung auf Basis von ASP.NET Core 2.1 mit Authentifizierungsfunktionen. Wie Sie eine Anwendung direkt bei der Erstellung mit der Authentifizierungsbibliothek ausstatten, erfahren Sie im Artikel **Authentifizierung unter ASP.NET Core**. Wie Sie eine vorhandene Anwendung mit den Authentifizierungsfunktionen nachrüsten, lesen Sie unter dem Titel **ASP.NET Core: Authentifizierung nachrüsten**. In beiden Fällen nutzen Sie gegebenenfalls die in der Bibliothek **Microsoft.AspNetCore.Identity.EntityFrameworkCore** enthaltenen Funktionen. Da wir im vorliegenden Artikel die Benutzeroberfläche sowie den dahinter steckenden Code anpassen wollen, müssen Sie die in der Bibliothek enthaltenen Dateien gegebenenfalls überschreiben. Wie das gelingt, lesen Sie im Artikel **ASP.NET Core: Authentifizierungsseiten anpassen** nach. Wir bauen im vorliegenden Artikel auf der Beispiellösung dieses Artikels auf.

Aktueller Stand

Im Standardzustand der Authentifizierung für ASP.NET Core-Webanwendungen sieht der bereits mit deutschen Texten versehene Registrierungsdialog wie in Bild 1 aus.

Wenn wir einen Blick in die Tabelle werfen, welche nach der Registrierung die daten des neuen Benutzers aufnehmen sollen, finden wir dort auch keine weiteren Felder, die man mit zusätzlichen Daten füllen könnte. Diese Tabelle finden Sie vor, wenn Sie **Ansicht|SQL Server-Objekt-Explorer** den entsprechenden Explorer einblenden und dort zur Tabelle **AspNetUsers** der passenden Datenbank navigieren und diesen Eintrag doppelt anklicken (siehe Bild 2).

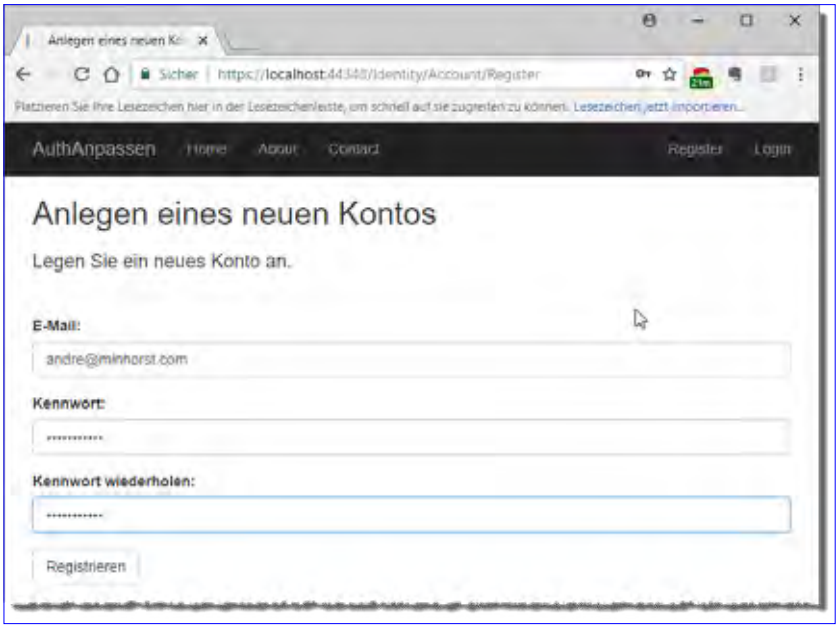


Bild 1: Dialog zum Anlegen eines Kontos vor dem Hinzufügen von Erweiterungen

Es gibt also zumindest schon einmal zwei Stellen, an denen wir Hand anlegen müssen, wenn wir beim Registrieren

eines neuen Benutzers mehr als nur die E-Mail-Adresse und das Kennwort (in verschlüsselter Form) speichern sollen.

Es gibt allerdings noch weitere Stellen – zum Beispiel das Model für den Benutzer in Form der Klasse **ApplicationUser** im Ordner **Data** sowie die passenden Elemente in der Code behind-Datei der Klasse **Register.cshtml**.

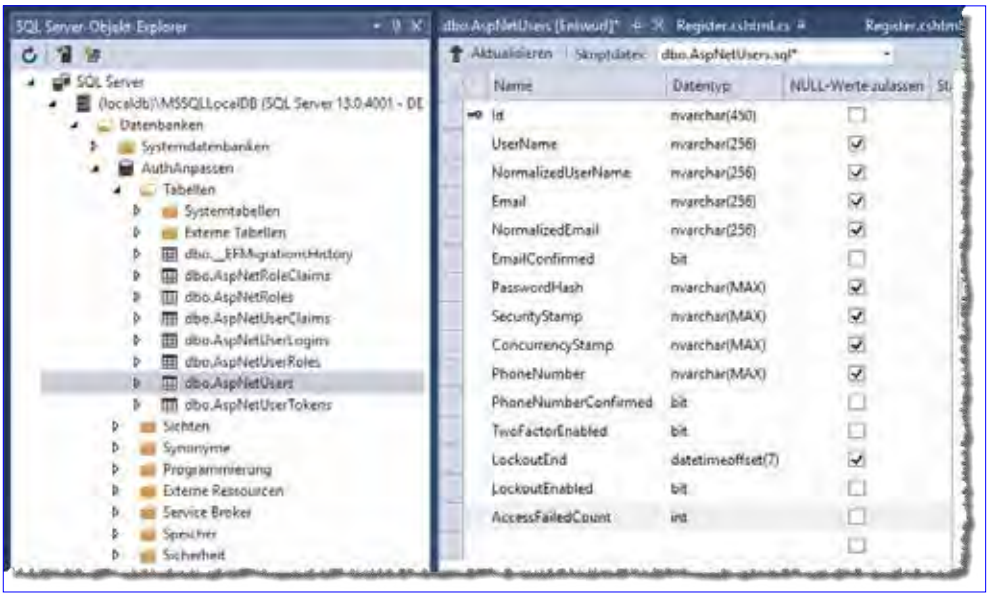


Bild 2: Datenmodell der Tabelle zum Speichern der Benutzerdaten

Schließlich wollen wir dem Benutzer auch noch die Möglichkeit geben, die bei der Registrierung angegebenen Daten zu aktualisieren, weshalb wir auch noch die Datei **ManageIndex.cshtml** erweitern müssen.

Schritt 1: Model anpassen

Damit wir bei den folgenden Schritten, also etwa beim Erweitern der Benutzeroberfläche, per IntelliSense auf die neuen Elemente zugreifen können, beginnen wir beim Model mit den Änderungen. Die dort verwendete Klasse **ApplicationUser** erbt von der Klasse **IdentityUser** und benötigte bisher noch keine eigenen Felder, da diese alle in **IdentityUser** steckten. Nun fügen wir die drei Felder **FirstName**, **LastName** und **BirthDate**hinzu:

```
01.     namespace AuthAnpassen.Data {
02.         public class ApplicationUser : IdentityUser {
03.             public string FirstName { get; set; }
04.             public string LastName { get; set; }
05.             public DateTime BirthDate{ get; set; }
06.         }
07.     }
```

Schritt 2: Vom Model zur Datenbank

Diese wollen wir nun zunächst auch in die Datenbank übertragen. Dazu benötigen wir die Paket-Manager-Konsole, die Sie, falls noch nicht sichtbar, über den Menübefehl **Extras|NuGet-Paket-Manager|Paket-Manager-Konsole** aktivieren. Wenn Sie noch keine Datenbank auf Basis der Klassen **ApplicationDbContext.cs** und **ApplicationUser.cs** erstellt haben, sind noch ein paar vorbereitende Schritte nötig. Der erste ist das Anpassen/Einfügen der Verbindungszeichenfolge in der Datei **appsettings.json**:

```
01.     {
02.       "ConnectionStrings": {
```

```
03.         "DefaultConnection": "Server=(localdb)\\mssqllocaldb;Database=AuthAnpassen;Trusted_Connection=True;
                                                                    MultipleActiveResultSets=true"
04.       }, ...
05.     }
```

Dann erstellen Sie die Datenbank mit den folgenden beiden Anweisungen:

```
Add-Migration Init
Update-Database
```

Hinweis: Wenn Sie die Datenbank erst jetzt erstellen, müssen Sie die folgenden Schritte nicht mehr ausführen, denn die Felder **FirstName**, **LastName** und **Birthday** wurden nun schon hinzugefügt. Wenn die Datenbank hingegen schon vorhanden ist, sind die folgenden beiden Anweisungen noch notwendig.

Zum Aktualisieren der Datenbank setzen Sie die folgenden zwei Befehle ab. Der erste fügt eine neue Klasse mit den für die Migration der Elemente in die Datenbank notwendigen Befehlen zum Projekt hinzu und nennt diese Klasse **AddFieldsAspNetUsers**:

```
Add-Migration AddFieldsAspNetUsers
```

Die Klasse hat den folgenden Code:

```
01.     using System;
02.     using Microsoft.EntityFrameworkCore.Migrations;
03.     namespace AuthAnpassen.Migrations {
04.         public partial class AddFieldsAspNetUsers : Migration {
05.             protected override void Up(MigrationBuilder migrationBuilder) {
06.                 migrationBuilder.AddColumn<DateTime>(
07.                     name: "Birthdate",
08.                     table: "AspNetUsers",
09.                     nullable: false,
10.                     defaultValue: new DateTime(1, 1, 1, 0, 0, 0, 0, DateTimeKind.Unspecified));
11.                 migrationBuilder.AddColumn<string>(
12.                     name: "FirstName",
13.                     table: "AspNetUsers",
14.                     nullable: true);
15.                 migrationBuilder.AddColumn<string>(
16.                     name: "LastName",
17.                     table: "AspNetUsers",
18.                     nullable: true);
19.             }
```

```
20.             protected override void Down(MigrationBuilder migrationBuilder) {
21.                 migrationBuilder.DropColumn(
22.                     name: "Birthdate",
23.                     table: "AspNetUsers");
24.                 migrationBuilder.DropColumn(
25.                     name: "FirstName",
26.                     table: "AspNetUsers");
27.                 migrationBuilder.DropColumn(
28.                     name: "LastName",
29.                     table: "AspNetUsers");
30.             }
31.         }
32.     }
```

Sie löscht also eventuell vorhandene Felder namens **BirthDate**, **FirstName** und **LastName** und erstellt diese unter den angegebenen Datentypen erneut. Dabei verwendet sie die **DropColumn**- beziehungsweise **AddColumn**-Felder des **MigrationBuilder**-Objekts. Nun rufen wir in der Paket-Manager-Konsole die folgende Anweisung auf:

```
Update-Database
```

Dies durchläuft die im Verzeichnis **Migrations** enthaltenen Klassen und führt diese aus. Dadurch landen nun auch die drei neuen Felder für die Klasse **ApplicationUser.cs** in der Tabelle **AspNetUsers** – siehe Bild 3.

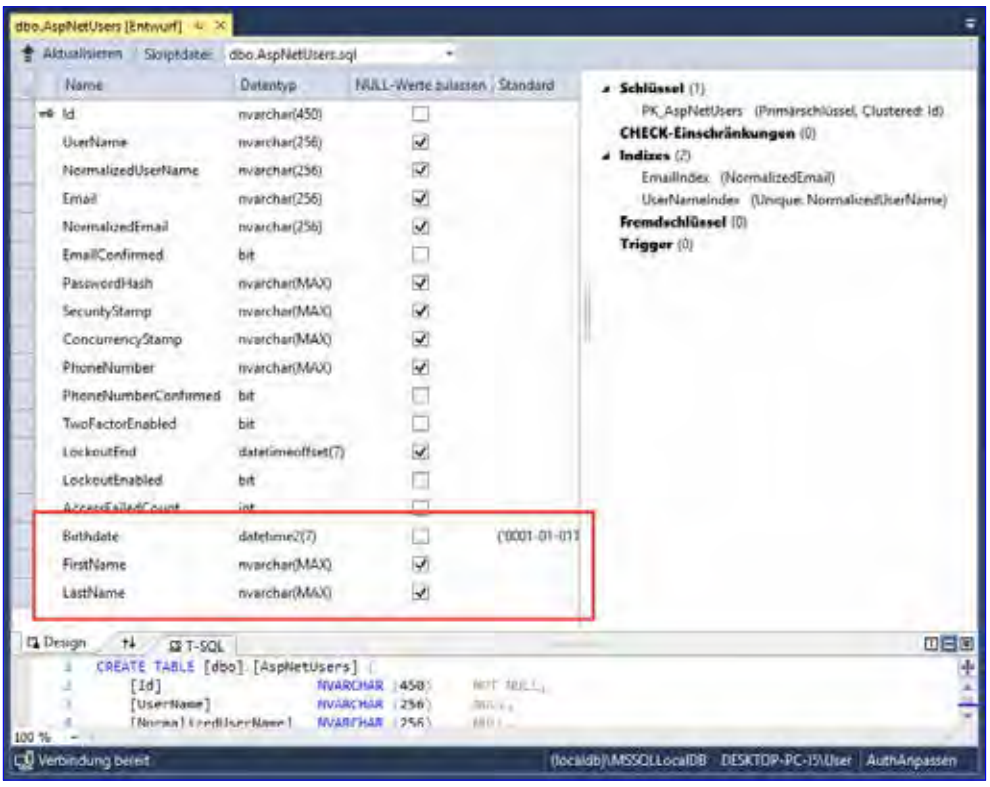


Bild 3: Die Tabelle **AspNetUsers** mit den drei neuen Feldern

Damit haben wir die Felder nun in der Klasse und in der zugrundeliegenden Tabelle angelegt. Damit diese auch über die Benutzeroberfläche verfügbar sind, sind noch einige weitere Schritte nötig.

Neue Felder zu den Webseiten hinzufügen

Wo genau benötigen wir die neuen Felder? Zunächst ein-

mal bei der Seite zum Registrieren eines neuen Benutzers, also auf **Areas|Identity|Pages|Account|Register.cshtml**. Außerdem gibt es die Möglichkeit, nach der Anmeldung das eigene Konto zu bearbeiten.

Dies geschieht auf der Seite **Areas|Identity|Pages|Account|Manage|Index.cshtml**. Natürlich können wir die vom Benutzer eingegebenen Daten auch noch an anderen Stellen anzeigen – beispielsweise im Kopfbereich nach der Anmeldung. Dies lassen wir allerdings vorerst weg.

Als Erstes passen wir in der Code behind-Datei **Register.cshtml.cs** die Klasse **InputModel** an. Diese erweitern wir um die beiden Eigenschaften **FirstName**, **LastName** und **BirthDate**:

```
01.     public class InputModel {
02.         [Required]
03.         [EmailAddress]
04.         [Display(Name = "E-Mail:")]
05.         public string Email { get; set; }
06.
07.         [Required]
08.         [StringLength(100, ErrorMessage = "The {0} must be at least {2} and at max {1} characters long.", MinimumLength = 6)]
09.         [DataType(DataType.Password)]
10.         [Display(Name = "Kennwort:")]
11.         public string Password { get; set; }
12.
13.         [DataType(DataType.Password)]
14.         [Display(Name = "Kennwort wiederholen:")]
15.         [Compare("Password", ErrorMessage = "Die beiden Kennwörter müssen identisch sein.")]
16.         public string ConfirmPassword { get; set; }
17.
18.         [Required]
19.         [Display(Name = "Vorname:")]
20.         public string FirstName { get; set; }
21.
22.         [Required]
23.         [Display(Name = "Nachname:")]
24.         public string LastName { get; set; }
25.
26.         [Required]
27.         [Display(Name = "Geburtsdatum:")]
28.         [DataType(DataType.Date)]
29.         [DisplayFormat(DataFormatString = "{0:dd.MM.yyyy}")]
30.         public DateTime Birthdate { get; set; }
31.     }
```

Razor-Pages: Daten aus Lookup-Tabellen

Im Artikel »Razor Pages mit Datenbankanbindung« haben wir gezeigt, wie Sie Daten aus einfachen Tabellen in einer Übersichtliste anzeigen, Details bearbeiten, neue Datensätze anlegen und vorhandene Datensätze ändern. In diesem Artikel nun wollen wir eine Lookup-Tabelle hinzunehmen, mit der wir die Anreden der Kunden aus der bereits vorhandenen Tabelle abbilden. Diese sollen dann in den Details per Nachschlagefeld ausgewählt werden. Eine kleine Schaltfläche neben dem Nachschlagefeld soll es dann ermöglichen, die Lookup-Daten zu bearbeiten.

Lookup-Entität hinzufügen

Im Artikel **EDM: 1:n-Beziehungen per Code First** erfahren Sie im Detail, wie Sie 1:n-Beziehungen zwischen zwei Entitäten aufbauen.

Für den vorliegenden Artikel wollen wir der bisher verwendeten Klasse **Customer** eine weitere Klasse namens **Salutation** hinzufügen, die wir dazu im Ordner **Models** anlegen:

```
public class Salutation {
    public int ID { get; set; }
    public string Name { get; set; }
    public ICollection<Customer> Customers { get; set; }
}
```

Um die Beziehung zwischen den beiden Klassen herzustellen, fügen wir der Klasse **Customer** auch noch zwei Elemente hinzu:

```
public class Customer {
    public int ID { get; set; }
    public string FirstName { get; set; }
    public string LastName { get; set; }
    public string Street { get; set; }
    public string Zip { get; set; }
    public string City { get; set; }
    public int SalutationID { get; set; }
    public Salutation Salutation { get; set; }
}
```

Schließlich fügen wir die neue Klasse noch als **DbSet** zur Klasse **CustomerManagementContext** im Verzeichnis **Data** hinzu:

```
public class CustomerManagementContext : DbContext {
    ...
```

```
    public DbSet<Customer> Customers { get; set; }
    public DbSet<Salutation> Salutations { get; set; }
    ...
}
```

Danach übernehmen Sie die Änderungen durch das Absetzen der folgenden beiden Anweisungen in der Paket-Manager-Konsole in die Datenbank, die in der Verbindungszeichenfolge der Datei **appsettings.json** angegeben ist:

```
Add-Migration Init
Update-Database
```

Um die Datenbank frisch anzulegen, löschen Sie die Datenbank zuvor aus dem SQL Server-Objekt-Explorer und entfernen auch alle Einträge im Verzeichnis **Migrations** im Projektmappen-Explorer.

Daten hinzufügen

Und damit beim Starten der Anwendung und beim Initialisieren auch noch ein paar Daten in der Tabelle **Salutations** landen und das Feld **Salutation** der Datensätze der Tabelle **Customers** mit Werten aus der Tabelle **Salutations** gefüllt wird, passen wir auch noch die Methode **Initialize** in der Klasse **DbInitializer** an:

```
public class DbInitializer {
    public static void Initialize(CustomerManagementContext dbContext) {
        dbContext.Database.EnsureCreated();
        if (dbContext.Customers.Any()) {
            return;
        }
        var salutations = new Salutation[] {
            new Salutation { Name = "Herr" },
            new Salutation { Name = "Frau" },
            new Salutation { Name = "Firma" }
        };
        foreach (Salutation s in salutations)            {
            dbContext.Salutations.Add(s);
        }
        var customers = new Customer[] {
            new Customer{FirstName="André", LastName="Minhorst", Street="Borkhofer Str. 17", Zip="47137",
                City="Duisburg", Salutation=salutations[0] }, ...
        };
        foreach (Customer c in customers) {
            dbContext.Customers.Add(c);
        }
        dbContext.SaveChanges();
```

```
    }
}
```

Diese legt nun zuerst einige Anreden an, bevor die Kunden mit dem jeweiligen Verweis auf die Anrede mit dem namens **Herr** hinzugefügt werden. Danach speichert die Methode die hinzugefügten Elemente mit der **SaveChanges**-Methode.

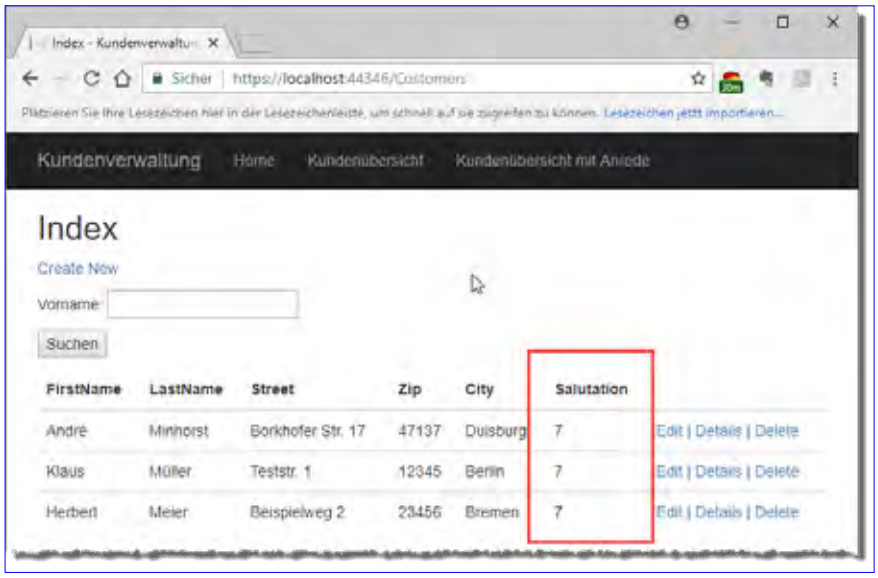


Bild 1: Ein erster Schritt – die Anzeige des Fremdschlüsselwertes

Anzeigen der Anreden in der Übersicht der Kunden

Als Erstes wollen wir die Seite **Pages\Customers\Index.cshtml** so anpassen, dass sie die Anrede der Kunden anzeigt. Dazu fügen wir der Tabelle auf der Seite **Index.cshtml** jeweils eine Spalte zum Element **theadItr** und zum Element **tbodyItr** hinzu. Dabei geben wir als Wert zunächst das Feld **SalutationID** an:

```
01.     <table class="table">
02.         <thead>
03.             <tr>
04.                 ... andere Überschriften ...
05.                 <th>
06.                     @Html.DisplayNameFor(model => model.Customer[0].Salutation)
07.                 </th>
08.                 <th></th>
09.             </tr>
10.         </thead>
11.         <tbody>
12.     @foreach (var item in Model.Customer) {
13.             <tr>
14.                 ... andere Feldinhalte ...
15.                 <td>
16.                     @Html.DisplayFor(modelItem => item.SalutationID)
17.                 </td>
18.                 ...
19.             </tr>
20.     }
21.         </tbody>
22.     </table>
```

Die Übersichtsseite zeigt nun eine neue Spalte mit dem Fremdschlüsselwert des Feldes **SalutationID** an (siehe Bild 1). Das ist zwar ein erster Schritt, aber nicht das, was wir erreichen wollen – wir möchten ja den Wert des Feldes **Name** des Datensatzes der Tabelle **Salutations**, der mit dem aktuellen Datensatz der Tabelle **Customers** verknüpft ist, anzeigen. Dazu ändern wir schlicht und einfach das Feld:

```
01.     <td>
02.         @Html.DisplayFor(modelItem => item.Salutation.Name)
03.     </td>
```

Dies führt allerdings nicht zum gewünschten Ergebnis – auf diese Weise bleibt die Spalte schlicht leer. Auch die einfache Verwendung von **item.Salutation** liefert eine leere Spalte. Allerdings wird auch keine Ausnahme ausgelöst – wo liegt also das Problem? Es liegt daran, dass wir nur die Daten der Tabelle **Customers** laden, aber nicht die der verknüpften Tabelle **Salutations** – und diese beherbergt nun einmal das Feld **Name**, mit dem wir die Anrede in der Tabelle ausgeben wollen.

Um diese Daten zu laden, gibt es verschiedene Möglichkeiten. Die einfachste ist diese hier, bei der wir einfach die Daten des **Salutation**-Elements zum jeweiligen **Customer**-Objekt auslesen, und zwar indem wir mit der **Include**-Methode arbeiten (mittlere Anweisung):

```
public async Task OnGetAsync(string firstName, string firstNamePart) {
    var customers = from m in _context.Customers select m;
    customers = customers.Include(c => c.Salutation);
    Customer = await customers.ToListAsync();
}
```

Dies liefert endlich die gewünschten Anreden in der Auflistung der Kunden (siehe Bild 2).

Anrede beim Anlegen oder Ändern auswählen

Nun gehen wir einen Schritt weiter und schauen uns den Dialog zum Bearbeiten eines Kunden an. Hier wollen wir das Steuerelement für die Auswahl der Anrede ebenfalls integrieren.

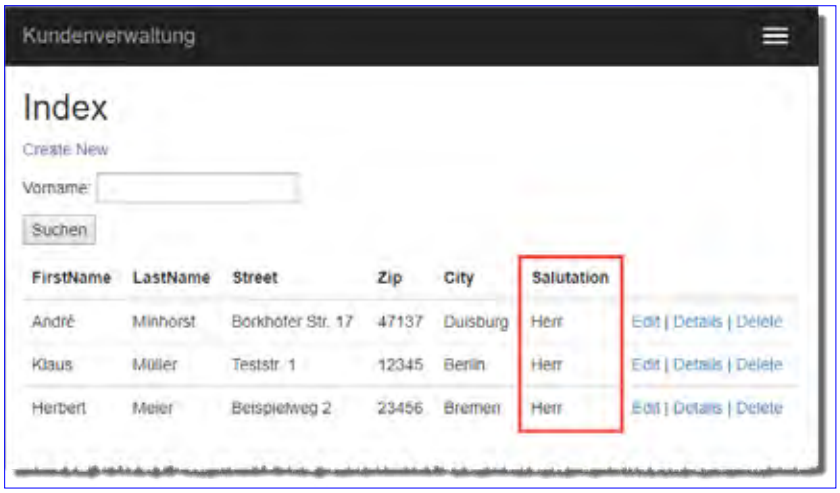


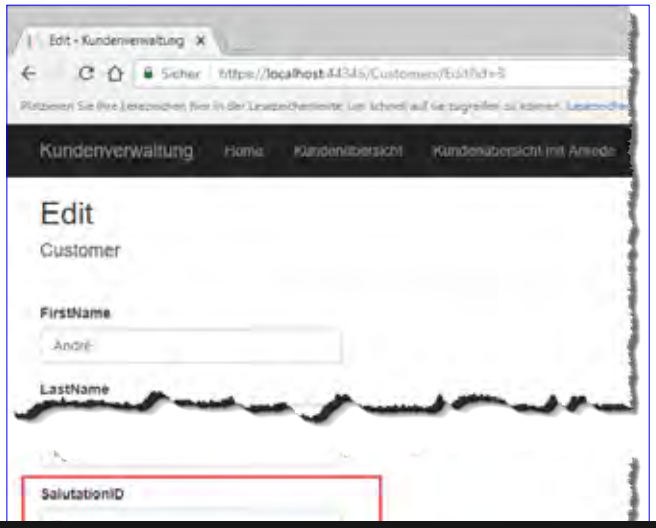
Bild 2: Kundenliste mit Anreden

In diesem Fall benötigen wir kein einfaches Textfeld, sondern ein Auswahlfeld. Unter HTML heißen diese Steuerelemente **Select**-Felder.

In diesem Fall müssen wir die Seite **Edit.cshtml** bearbeiten. Wir beginnen wieder einfach und fügen dem Formular mit den

Steuerelementen zunächst ein Steuerelement zur Eingabe des Fremdschlüsselfeldes **SalutationID** hinzu. Dieses statten wir auch mit einer Beschriftung und einer Validierung aus wie die übrigen Steuerelemente:

```
01.     <form method="post">
02.         <div asp-validation-summary="ModelOnly" class="text-danger"></div>
03.         <input type="hidden" asp-for="Customer.ID" />
04.         ...
05.         <div class="form-group">
06.             <label asp-for="Customer.SalutationID" class="control-label"></label>
07.             <input asp-for="Customer.SalutationID" class="form-control" />
08.             <span asp-validation-for="Customer.SalutationID" class="text-danger"></span>
09.         </div>
10.         <div class="form-group">
11.             <input type="submit" value="Save" class="btn btn-default" />
12.         </div>
13.     </form>
```



Damit erreichen wir unser Zwischenziel – siehe Bild 3. Allerdings wollen wir dem Benutzer nicht zumuten, dass er die Zahlenwerte für die jeweiligen Anreden von Hand in das Formular eingeben muss. Stattdessen wollen wir ein Auswahlfeld implementieren. Dazu sind nur zwei kleine Änderungen notwendig.

Als Erstes müssen wir die Daten für das Auswahlfeld in der Code behind-Datei **Edit.cshtml.cs** bereitstellen, und zwar in der Methode **OnGetAsync**. Dieser fügen wir als letzte Zeile vor der Anweisung **return Page()** eine Anweisung hinzu,

Die Leseprobe dieses Artikels ist hier zu Ende.
Willst Du mehr?
ZUM SHOP
2.400 Seiten Know-how rund um die Datenbankentwicklung mit .NET
KOMPENDIUM
DATENBANK ENTWICKLER
KOMPENDIUM
André Minhorst Verlag
Spare 20 EUR mit Code dbew20

AUSGABE 05/2018

DATENBANK ENTWICKLER

MAGAZIN FÜR DIE DATENBANKENTWICKLUNG MIT VISUAL STUDIO FÜR DESKTOP, WEB UND CO.

TOP-THEMEN:

GRUNDLAGEN	Von Access zum Entity Framework: Basics	SEITE 3
DEPLOYMENT	Datenbank-Initialisierung	SEITE 6
MIGRATION	Entity Framework: Datenbankmigration	SEITE 16
ACCESS ZU .NET	Von Access zu Entity Framework: Datenmodell	SEITE 43
ACCESS ZU .NET	EDM: DataGrid als Datenblatt	SEITE 57

André Minhorst Verlag

Von Access zum Entity Framework: Basics

In den vorherigen Ausgaben haben wir bereits verschiedene Techniken erläutert und unter anderem kleine Desktop- und Webanwendungen programmiert. Dabei sind wir ein wenig vom eigentlichen Ziel des Magazins abgewichen – Access-Entwicklern die Möglichkeiten von Visual Studio und den dortigen Technologien für die Migration von Access-Anwendungen in Desktop- oder Webanwendungen aufzuzeigen. Mit diesem Artikel kehren wir dorthin zurück und erklären, wie Sie den Umzug einer Anwendung von Access zu .NET einleiten und welche Techniken wir dazu in Zukunft nutzen wollen.

Es gibt zahllose Möglichkeiten, wie Sie Desktop-Anwendungen ähnlich denen, die man mit Access erstellt, unter .NET zu programmieren. Das beginnt mit der Wahl der Technik für die Gestaltung der Benutzeroberfläche, geht über die Entscheidung, welches Datenbanksystem man einsetzt bis hin zur Technik für den Datenzugriff.

Benutzeroberfläche

Was die Gestaltung der Benutzeroberfläche angeht, haben wir von der ersten Ausgabe an gezeigt, wie Sie die Benutzeroberfläche mit WPF umsetzen. In der Zwischenzeit haben wir auch einige Artikel darauf verwendet, die Erstellung von Web-Anwendungen auf Basis von HTML zu erläutern.

Backend

Bezüglich des verwendeten Backends haben wir verschiedene Phasen durchlaufen: Zu Beginn haben wir gezeigt, wie Sie über ADO.NET auf die Tabellen einer Access-Datenbank zugreifen können. Access-Datenbanken sind aber aus verschiedenen Gründen nicht die erste Wahl für das Backend: Erstens ist die Dateigröße auf zwei Gigabyte beschränkt, zweitens gibt es kein Berechtigungssystem und drittens kann es nur eine stark begrenzte Anzahl gleichzeitiger Zugriffe verarbeiten. Hier sind andere Lösungen gefragt, weshalb wir auf den SQL Server respektive dessen abgespeckter Version namens LocalDb umgestiegen sind. Für solche Zwecke, wo kein SQL Server vorhanden ist, haben wir den Einsatz von SQLite erläutert.

Datenzugriffstechnik

Um den Umstieg zu erleichtern, haben wir zu Beginn noch auf Basis von Access-Datenbanken gearbeitet und auf diese über ADO.NET zugegriffen. Aus den oben genannten Gründen haben wir uns dann auf SQL Server und SQLite konzentriert. Das hat noch einen anderen Grund: Wer die Vorteile der objektorientierten Entwicklung stärker nutzen möchte und gleichzeitig weniger Zeit in die Programmierung des Datenzugriffs stecken möchte, kann einen sogenannten O/R-Mapper nutzen, einen Mapper, der ein Bindeglied zwischen der objektorientierten und der relationalen Welt bildet. In diesem Fall wollen wir den Zugriff auf die Daten über eine Reihe geeigneter Objekte realisieren, indem wir zum Beispiel auf die in Tabellen gespeicherten Entitäten in Form von Objekten zugreifen – und auf die Gesamtheit dieser Entitäten über entsprechende Auflistungsklassen.

Unsere Wahl fiel dabei auf das von Microsoft entwickelte Entity Framework. Hier haben wir zu Beginn erläutert, wie Sie ein Entity Data Model, also im Prinzip die Abbildung des relationalen Datenmodells, auf Basis einer bestehenden Tabelle erstellen. Dazu ist, wenn Sie von Access aus kommen, zuerst die Migration der Datenbank in eine Datenbank eines geeigneten Datenbanksystems erforderlich. Das Entity Framework bietet Treiber für verschiedene Datenbanksysteme an, zum Beispiel SQLite oder den

SQL Server, aber nicht für Access – daher können wir hier nicht bei Access als Backendsystem verharren. Der Einsatz des Entity Frameworks bedingt also die komplette Abkehr von Access.

Wann Access nutzen und wann nicht

Nebenbei bemerkt: Es ist für viele Szenarien sinnvoll, mit Access zu arbeiten – also etwa dann, wenn nicht sehr viele Zugriffe auf die Datenbank erfolgen und kein Berechtigungssystem erforderlich ist. Die Geschwindigkeit, mit der Sie eine Anwendung auf Basis von Access erstellen, dürfte mit keinem anderen System möglich sein, auch nicht mit Visual Studio.

Es ist auch dann noch sinnvoll, mit Access zu arbeiten, wenn es nur das Frontend für den Zugriff auf eine Backend-Datenbank etwa auf dem SQL Server stellt. Auch dann profitieren Sie vom einfachen Entwurf der Benutzeroberfläche. Wenn es jedoch darum geht, ein Backend zu einem alternativen Frontend zu Access zu wählen, also etwa für eine WPF- oder Webanwendung, dann sollten Sie nicht auf den Tabellen in einem Access-Backend verharren, sondern eine andere Lösung wählen.

Von Database First zu Model First

In den ersten Artikeln zum Thema Entity Framework haben wir dann auch noch die Tabellen in einer SQL Server- oder SQLite-Datenbank erstellt und dann ein Entity Data Model auf Basis dieser Tabellen erstellen lassen. Dazu haben wir dann meist die Vorlage EF Designer aus Datenbank verwendet. Dieser Ansatz bot uns auch noch die Sicherheit, dass wir das so generierte Modell in einer grafischen Benutzeroberfläche anpassen konnten.

Hinter dieser grafischen Benutzeroberfläche steckte dabei das Entity Data Model, das aus den Entitäten beziehungsweise Entitätsklassen besteht sowie einer Auflistungsklasse für jede Entität. In den Entitäten sind dann die Beziehungen zwischen den Entitäten durch entsprechende Eigenschaften definiert.

In den vorherigen Ausgaben sind wir dann einen Schritt weitergegangen und haben begonnen, uns für eine Weile von dem Vorhandensein einer fertigen Datenbank als Ausgangspunkt zu lösen. Dort haben wir zuerst die Entitätsklassen erstellt und die Beziehungen zwischen den Klassen über die dazu vorgesehenen Eigenschaften festgelegt. Daraus haben wir dann auf umgekehrtem Wege wie zuvor die Datenbank erstellt. Dieser Ansatz heißt Code First.

Von Access über Model First zur Datenbank

In dieser Ausgabe von **DATENBANKENTWICKLER** gehen wir schließlich noch einen Schritt weiter und wollen ein Entity Data Modell aus einer Access-Datenbank entwickeln. Dazu haben wir im Artikel **Von Access zu Entity Framework: Datenmodell** beschrieben, wie Sie mit einer VBA-Routine aus Access heraus den Code generieren können, aus dem das Entity Data Model besteht – also aus den Entitätsklassen und den Anweisungen zur Deklaration von **DbSet**-Objekten für den Zugriff auf die Auflistungen. Ein Objekt auf Basis einer Entitätsklasse entspricht dabei einem Datensatz einer Tabelle und ein **DbSet**-Objekt ist die Grundlage, Auflistungsklassen auf Basis der Daten einer Tabelle zu erzeugen, also quasi mit Recordsets vergleichbar ist.

Das so erstellte Entity Data Model können Sie dann nutzen, um mit wenigen Anweisungen eine Datenbank auf Basis der Definition der Klassen zu erzeugen.

Im Artikel **Von Access zu Entity Framework: Daten** berichten wir dann darüber, wie Sie Anweisungen zum Füllen der zuvor auf Basis des Entity Data Models erzeugten Klassen zusammenstellen. Dabei durchlaufen wir die Tabellen, deren Definition

Entity Framework: Datenbankinitialisierung

Wenn Sie mit Code First arbeiten, also Ihre Datenbank auf Basis eines Entity Data Models erstellt wird, können Sie verschiedene Strategien auswählen, um die Datenbank zu erstellen oder anzupassen, wenn Sie die Anwendung an einen anderen Benutzer weitergeben. Dieser Artikel zeigt, welche Möglichkeiten es gibt und wie diese funktionieren.

In den letzten Ausgaben von **DATENBANKENTWICKLER** haben wir vermehrt mit Anwendungen gearbeitet, in denen wir ein Entity Data Model definiert haben, also verschiedene Klassen für die Objekte und entsprechende Auflistungen. Daraus erstellt das Entity Framework dann bei Bedarf eine Datenbank, welche die Klassen und Auflistungen berücksichtigt. Dabei gibt es verschiedene Möglichkeiten, die Datenbank zu erstellen, wenn diese noch nicht vorhanden ist oder diese anzupassen, wenn Sie eine neue Version der Datenbank an den Benutzer weitergeben.

Als Voraussetzung wollen wir ein einfaches Modell heranziehen, dass nur aus den beiden Tabellen **Articles** und **Categories** besteht. Diese haben wir in einem neuen Projekt namens **InitializationSample** als Visual Basic-WPF-App angelegt, dem wir ein neues Objekt des Typs **ADO.NET Entity Data Model** für ein **Leeres Code First-Modell** namens **ArticleContext** hinzugefügt haben.

Hier legen wir im Unterordner **Data** die erste Klasse namens **Articles.vb** an:

```
Public Class Article
    Public Property ID As Integer
    Public Property Name As String
    Public Property CategoryID As Integer
    Public Property Category As Category
    Public Property Price As Decimal
End Class
```

Die zweite Klasse nimmt die Kategorien auf und heißt **Categories.cs**:

```
Public Class Category
    Public Property ID As Integer
    Public Property Name As String
    Public Property Articles As ICollection(Of Article)
End Class
```

Dazu kommen zwei Zeilen, welche die Auflistungen der Elemente dieser beiden Klassen repräsentieren. Diese fügen wir in der Klasse ein, die beim Hinzufügen des Entity Data Models angelegt wurde und den Namen **ArticleContext.vb** trägt. Diese sieht dann so aus:

```
Imports System.Data.Entity
```

```
Public Class ArticleContext
    Inherits DbContext
    Public Sub New()
        MyBase.New("name=ArticleContext")
    End Sub
    Public Property Categories() As DbSet(Of Category)
    Public Property Articles() As DbSet(Of Article)
End Class
```

Datenbank beim ersten Zugriff erstellen

Entity Framework hat einen Mechanismus, der dafür sorgt, dass eine Datenbank automatisch erstellt wird, wenn Sie das erste Mal auf diese zugreifen, aber noch keine Datenbank vorhanden ist. Um dies auszuprobieren, fügen Sie der Code behind-Klasse des Fensters **MainWindow.xaml** zunächst den Konstruktor hinzu und füllen diesen dann mit den folgenden Anweisungen:

```
Class MainWindow
    Public Sub New()
        Using dbContext As ArticleContext = New ArticleContext
            Dim MyCategory As Category
            MyCategory = New Category()
            MyCategory.Name = "Getränke"
            dbContext.Categories.Add(MyCategory)
            dbContext.SaveChanges()
        End Using
    End Sub
End Class
```

Die Methode **New** erstellt einen neuen Kontext auf Basis der Klasse **ArticleContext** und speichert diese in der Variablen **dbContext**. Dann erstellen wir ein neues Objekt des Typs **Category**, weisen der Eigenschaft **Name** den Wert **Getränke** zu und fügen das neue Objekt zur Auflistung **Categories** des Objekts **dbContext** hinzu. Die Änderungen speichern wir dann mit der Methode **SaveChanges**.

Nun brauchen Sie die Anwendung nur noch zu starten. Es dauert dann ein wenig länger bis zum Erscheinen des Anwendungsfensters, aber dafür wird in der Zwischenzeit auch gleich die Datenbank erstellt.

Diese finden Sie dann nach dem Beenden der Anwendung unter Visual Studio im SQL Server-Objekt-Explorer, den Sie mit dem Menübefehl **Ansicht|SQL Server-Objekt-Explorer**

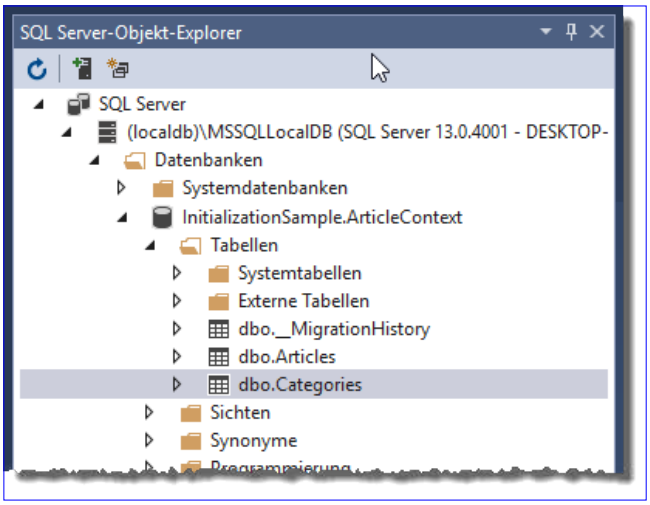


Bild 1: Die neue Datenbank im SQL Server-Objekt-Explorer

einblenden können. Hier klappen Sie den Eintrag **Datenbanken** auf und finden die Datenbank **InitializationSample.ArticleContext** vor (siehe Bild 1).

Wenn Sie nun noch das Kontextmenü des Eintrags **dbo.Categories** öffnen und dort den Befehl **Daten anzeigen** auswählen, erscheinen die Daten der Tabelle **Categories** – in diesem Fall der eine in der Konstruktor-Methode angelegte Datensatz mit dem Wert **Getränke** im Feld **Name**.

Ergebnisse beim Anlegen der Datenbank

Schauen wir uns an, was im Detail geschehen ist. Als Erstes ist festzuhalten: Es wurde eine Datenbank angelegt, und zwar auf dem Server **(LocalDb)\MSSQLLocalDB** und unter dem Namen **InitializationSample.ArticleContext**.

Wenn Sie diese Einstellung ändern wollen, können Sie das in der Datei **App.config** (für Desktop-Anwendungen) oder **Web.config** (für Web-Anwendungen) erledigen, wo Sie den folgenden Abschnitt anpassen müssen:

```
<connectionStrings>
    <add name="ArticleContext" connectionString="data source=(LocalDb)\MSSQLLocalDB;
        initial catalog=InitializationSample.ArticleContext;integrated security=True;MultipleActiveResultSets=True;
        App=EntityFramework" providerName="System.Data.SqlClient" />
</connectionStrings>
```

Woher weiß Entity Framework, dass es genau diese Verbindungszeichenfolge nutzen soll? Wir sehen hier, dass diese den Namen **ArticleContext** aufweist. Dieser wird in der Konstruktor-Methode der Klasse **ArticleContext.vb**, die mit dem Entity Data Model erstellt wurde, referenziert:

```
Public Sub New()
    MyBase.New("name=ArticleContext")
End Sub
```

Wenn Sie also eine andere Verbindungszeichenfolge nutzen wollen, stellen Sie diese hier ein.

Alternativer Datenbankname

Während wir dem **New**-Konstruktor von **MyBase** soeben den Namen der Verbindungszeichenfolge übergeben haben, können Sie hier auch den Namen der zu erstellenden Datenbank angeben. Dazu lassen Sie einfach den Namen des Parameters weg und geben direkt den gewünschten Datenbanknamen an, zum Beispiel **InitializationSample**:

```
Public Sub New()
    MyBase.New("InitializationSample")
End Sub
```

Sie können aber auch einfach den Namen der Datenbank in der Verbindungszeichenfolge anpassen. Diesen finden Sie unter dem Parameternamen **initial catalog**.

Tabellen erstellen

Welche Tabellen Entity Framework in der neuen Datenbank erstellen soll, entnimmt es ebenfalls der Klasse **ArticleContext.vb**. Hier berücksichtigt es einfach alle als **DbSet** deklarierten öffentlichen Eigenschaften, also aktuell diese hier:

```
Public Property Categories() As DbSet(Of Category)
Public Property Articles() As DbSet(Of Article)
```

Felder hinzufügen

Nun fehlen noch die Felder. Welche Entity Framework hier hinzufügen muss, entnimmt es der Definition der Klassen, die in den **DbSet**-Objekten angegeben sind (zum Beispiel **DbSet(Of Category)**).

Wie Sie die Felder anlegen müssen, damit beispielsweise Beziehungen zwischen den Tabellen in der Zieldatenbank entstehen, erfahren Sie in den Artikeln **EDM: 1:n-Beziehungen per Code First**, **EDM: 1:1-Beziehungen per Code First** und **EDM: m:n-Beziehungen per Code First**.

Der nächste Start

Wenn Sie die Anwendung nun nochmals öffnen, ist die Datenbank bereits vorhanden. Entity Framework muss diese dann nicht erneut anlegen und kann die vorhandene Datenbank verwenden.

Änderungen am Datenmodell

Wenn Sie nun eine Änderung am Datenmodell vornehmen, indem Sie etwa das Feld **Description** zur Klasse **Article** hinzufügen, wollen Sie diese Änderung natürlich auch auf das Datenmodell einer eventuell bereits existierenden Datenbank übertragen. Also probieren wir aus, was nach der folgenden Anpassung und einem Neustart der Anwendung geschieht:

```
Public Class Article
    ...
    Public Property Description As String
End Class
```

Das Ergebnis sehen Sie in Bild 2. Die Fehlermeldung ist eindeutig: Entity Framework hat erkannt, dass es seit dem Erstellen der Datenbank eine Änderung des Modells gegeben hat und schlägt den Einsatz von Code First-Migrationen zur Aktualisierung vor. Aber was bedeutet das? Darauf kommen wir später zurück.

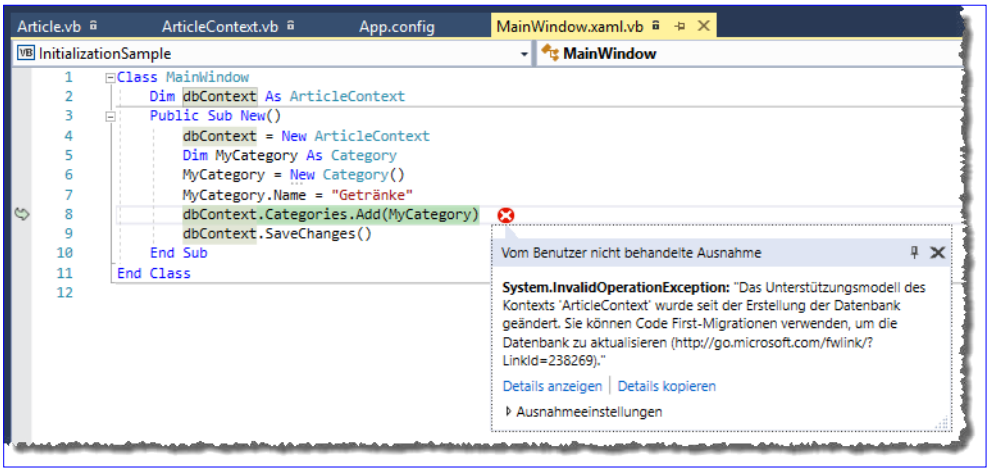


Bild 2: Fehler bei Unterschieden zwischen Entity Data Model und der Datenbank

Datenbank-Initialisierer

Zuerst einmal schauen wir uns an, was hier am Werk war, um unsere Datenbank initial beim ersten Zugriff auf die Datenbank zu erstellen. Dass dies passiert ist, liegt daran, dass standardmäßig ein Datenbank-Initialisierer namens **CreateDatabaseIfNotExists** verwendet wird. Dies ist einer der eingebauten Datenbank-Initialisierer. Die Namen und die Funktion der Datenbank-Initialisierer lauten:

- **CreateDatabaseIfNotExists**: Der Standard-Datenbank-Initialisierer. Die Datenbank wird nur dann erstellt, wenn Sie noch nicht vorhanden ist. Wenn sich das Modell ändert, löst dies beim Zugriff auf eine veraltete Version der Datenbank einen Fehler aus (siehe oben).

- **DropCreateDatabaseWhenModelChanges**: Auch dieser Initialisierer erstellt eine Datenbank, wenn diese noch nicht vorhanden ist. Zusätzlich löscht sie eine bestehende Datenbank und erstellt diese neu, wenn sich das Datenmodell in der Zwischenzeit geändert hat. Für den produktiven Betrieb ist diese Version nicht nützlich, da mit der Datenbank ja auch alle enthaltenen Daten gelöscht werden.

- **DropCreateDatabaseAlways**: Dieser Initialisierer löscht die Datenbank jedes Mal beim ersten Zugriff und erstellt diese auf Basis des aktuellen Modells neu.

- **MigrateDatabaseToLatestVersion**: Dieser Initialisierer führt offene Migrationen aus. Mehr zu dieser Option erfahren Sie im Artikel **Entity Framework: Datenbankmigration**.

Wie und wo stellen Sie einen Datenbank-Initialisierer ein, wenn Sie nicht die Standardeinstellung verwenden wollen? Dieser gehört in die Konstruktor-Methode der Kontext-Klasse – hier am Beispiel des Standard-Datenbank-Initialisierers:

```
Public Sub New()
    MyBase.New("name=ArticleContext")
    Database.SetInitializer(New CreateDatabaseIfNotExists(Of ArticleContext)())
End Sub
```

Wenn Sie etwa immer eine neue Datenbank erstellen wollen, verwenden Sie die folgende Anweisung:

```
Database.SetInitializer(New DropCreateDatabaseAlways(Of ArticleContext)())
```

Wenn Sie dies testen wollen, fügen Sie einfach der Tabelle **Categories** der Datenbank einen neuen Datensatz hinzu und erstellen die Datenbank dann neu.

Initialisierung manuell anstoßen

Sie können die Initialisierung auch mit einem geeigneten Befehl anstoßen, wenn Sie nicht warten wollen, bis der Benutzer erstmalig auf die Daten der Anwendung zugreift. In diesem Fall verwenden Sie folgende Variante:

```
Public Sub New()
    Using dbContext As ArticleContext = New ArticleContext
```

Entity Framework: Datenbankmigration

Unter Access hatten Sie ein Problem, wenn Sie eine neue Version einer Datenbank ausliefern wollten, deren Datenmodell sich geändert hat. Dann war Handarbeit angesagt! Das Entity Framework bietet für das Übertragen von Änderungen am Datenmodell die sogenannten Migrationen an. Dieser Artikel zeigt, was es sich damit auf sich hat und wann Sie Migrationen gegenüber Datenbankinitialisierern nutzen sollten.

Wenn Sie eine Datenbank weiterentwickeln, die schon im produktiven Betrieb genutzt wird, gibt es verschiedene Änderungen, die unterschiedlich gehandhabt werden. Wenn Sie lediglich Änderungen an der Benutzeroberfläche oder an der Anwendungslogik vornehmen, haben Sie leichtes Spiel: Sie brauchen dann einfach nur die neue Version der Anwendung zu verteilen, die dann auf die vorhandene Datenbank zugreift.

Interessant wird es, wenn Sie nicht nur Änderungen an der Benutzeroberfläche oder der Anwendungslogik vornehmen, sondern wenn Sie das Entity Data Model manipulieren. Dann müssen nämlich auch die Tabellen in der zugrunde liegenden Datenbank angepasst werden, da sonst das Modell nicht mehr zur Datenbank passt. Hier wird es interessant, denn Entity Framework bietet die sogenannten Migrations, um Änderungen an den Entities zuverlässig in die Datenbank zu übertragen.

Während der Entwicklung der Anwendung, also bevor diese erstmals auf dem Rechner des Benutzers landet, ist der Umgang mit Aktualisierungen im Entity Data Model einfach: Man konnte dann einfach die komplette vorhandene Datenbank löschen und diese auf Basis des Entity Data Models neu aufbauen. Gegebenenfalls hat man dann noch mit der **Seed**-Methode einige Daten zu den frisch erstellten Tabellen hinzugefügt. Wie das im Detail funktioniert, erfahren Sie im Artikel **Entity Framework: Datenbank-Initialisierer**.

Sobald die Anwendung aber beim Kunden im Einsatz ist und dieser damit begonnen hat, Daten in die Tabellen zu schreiben oder diese zu ändern, kann man nicht mehr so einfach die komplette Datenbank löschen und neu erstellen – dann wäre die ganze Arbeit des Benutzers verloren. Alternativen wären das manuelle Schreiben von SQL-Anweisungen, mit denen die Änderungen im Entity Data Model feingranular in die Datenbank übertragen werde, ohne diese löschen und neu erstellen zu müssen. Oder man erstellt die Datenbank neu und erzeugt ein Skript, mit dem die Daten aus der Datenbank in der alten Version in die neue Version übertragen werden. Auch das ist mit viel Arbeit verbunden. Aber schauen wir uns doch an, was Entity Framework zu diesem Thema zu bieten hat.

Beispielprojekt

Das Beispielprojekt **DatabaseMigrations** auf Basis der Vorlage **Visual BasicIWPF-App** haben wir wieder mit einem neuen Element des Typs **ADO.NET Entity Data Model** namens **ArticleContext** ausgestattet, für das wir als Modellinhalt **Leeres Code First-Modell** gewählt haben. Dem Projekt haben wir dann einen neuen Ordner namens **Data** mit den beiden Klassen **Article.cs** und **Category.cs** hinzugefügt. Außerdem haben wir der durch Entity Framework erstellten Klasse **ArticleContext.vb** noch die beiden **DbSet**-Auflistungen **Articles** und **Categories** hinzugefügt (siehe Beispielprojekt).

Migrations

Dieses bietet nämlich eine Technik namens **Migrations** an. Dabei gibt es zwei verschiedene Funktionen:

- Code-basierte Migration: Muss durch den Entwickler ausgelöst werden.
- Automatische Migration: Erfolgt automatisch, hat aber Einschränkungen.

In den folgenden Abschnitten schauen wir uns die verschiedenen Funktionen an.

Code-basierte Migration

Die code-basierte Migration erfordert den Aufruf verschiedener Befehle über die Paket-Manager-Konsole. Diese starten Sie über den Menübefehl **Extras|NuGet-Paket-Manager|Paket-Manager-Konsole**. Hier können Sie die folgenden Anweisungen eingeben:

- **Enable-Migrations**: Aktiviert die Migration für das aktuelle Projekt und erstellt einen neuen Ordner namens **Migrations** mit einer neuen Klasse namens **Configuration**.
- **Add-Migration**: Erfasst die Unterschiede zwischen dem Entity Data Model im Vergleich zur Datenbank und trägt diese in jeweils eine neue Klasse in zwei Methoden namens **Up** und **Down** ein. Die Methoden enthalten die Anweisungen zum Erstellen, Löschen oder Ändern der Datenbankobjekte entsprechend den Änderungen im Entity Data Model.
- **Update-Database**: Diese Methode überträgt alle noch nicht übertragenen Änderungen aus den mit **Add-Migration** erstellten Klassen in die Tabellen der Datenbank.

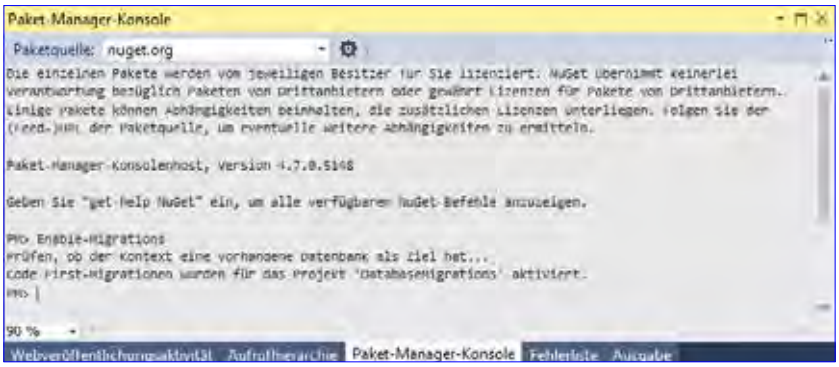


Bild 1: Aktivieren der Datenbank-Migrationen

Um die Migrations-Funktionen zu nutzen, müssen sie diese mit der ersten Anweisung **Enable-Migrations** aktivieren. Diese Anweisung setzen Sie in der Paket-Manager-Konsole ab (siehe Bild 1):

```
Enable-Migrations
```

Dies quittiert die Paket-Manager-Konsole damit, dass Code First-Migrationen für das Projekt aktiviert sind. Was hat sich dadurch verändert? Im Projektmappen-Explorer finden wir nun einen neuen Ordner namens **Migrations** mit der Klasse **Configurations.vb** (siehe Bild 2).

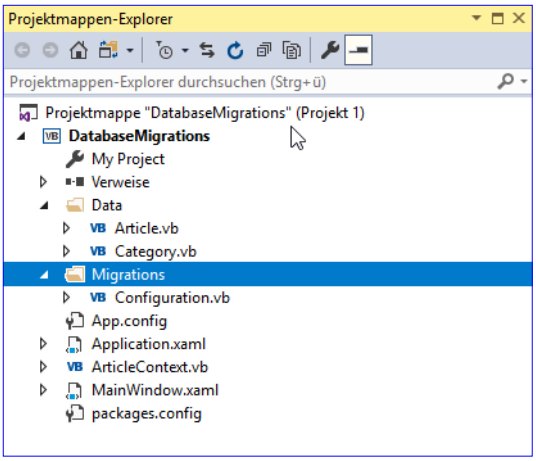


Bild 2: Neuer Ordner **Migrations**

Die Klasse stellt zwei Methoden bereit – die Konstruktor-Methode **New** sowie eine Methode namens **Seed**:

```
Imports System.Data.Entity.Migrations
```

```
Namespace Migrations
    Friend NotInheritable Class Configuration
        Inherits DbMigrationsConfiguration(Of ArticleContext)
        Public Sub New()
            AutomaticMigrationsEnabled = False
        End Sub
        Protected Overrides Sub Seed(context As ArticleContext)
            '  This method will be called after migrating to the latest version.

            '  You can use the DbSet(Of T).AddOrUpdate() helper extension method
            '  to avoid creating duplicate seed data.
        End Sub
    End Class
End Namespace
```

Die Konstruktor-Methode stellt den Wert der Variablen **AutomaticMigrationsEnabled** auf den Wert **False** ein. Was bedeutet das? Sie setzen **AutomaticMigrationsEnabled** auf **True**, wenn Sie keine Versionierung in dem Sinne planen, dass Sie nur Upgrades verwenden und keine Downgrades und keine strikte Versionierung planen. Das schauen wir uns weiter unten unter **Automatische Migration aktivieren** an. Wenn Sie die Variable auf dem Wert **False** belassen, müssen Sie die Migrationsschritte selbst definieren. Das schauen wir uns gleich im Anschluss an.

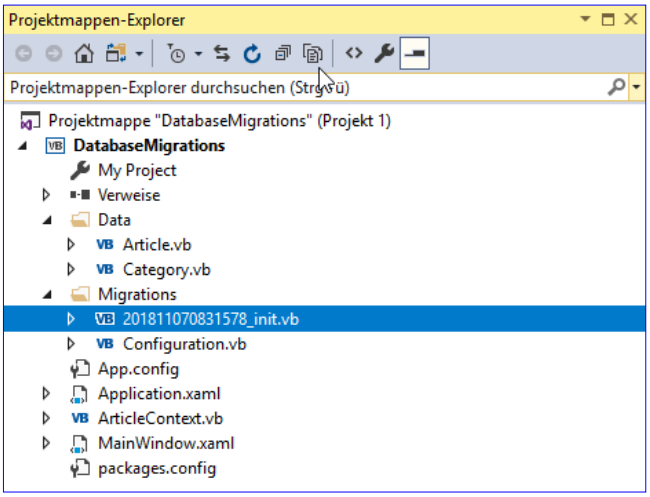


Bild 3: Neue Klasse, die auf **init.vb** endet

Änderungen in Klasse notieren

Danach können Sie erstmals die Anweisung **Add-Migration** aufrufen und dieser einen Parameter mitgeben, der später als Teil des Namens der durch diese Anweisung erstellten Klasse verwendet wird (der Rest besteht aus Datum und Zeit). Die erste Migration soll beispielsweise den Text **Init** im Namen der zu erstellenden Klasse tragen. Die folgende Anweisung setzen Sie ebenfalls in der Paket-Manager-Konsole ab. Die Ausführung kann einige Sekunden dauern:

```
Add-Migration Init
```

Die so erstellte Klasse heißt **201811070831578_init.vb** und taucht im Ordner **Migrations** des Projektmappen-Explorers auf (siehe Bild 3). Sie sieht mit den beiden Methoden **Up** und **Down** in gekürzter Form wie folgt aus:

```
Imports System.Data.Entity.Migrations
```

```
Namespace Migrations
    Public Partial Class init
        Inherits DbMigration
        Public Overrides Sub Up()
            CreateTable("dbo.Articles",
                Function(c) New With
                    {
                        .ID = c.Int(nullable := False, identity := True),
                        .Name = c.String(),
                        .CategoryID = c.Int(nullable := False),
                        .Price = c.Decimal(nullable := False, precision := 18, scale := 2),
                        .Description = c.String()
                    }) _
                .PrimaryKey(Function(t) t.ID) _
                .ForeignKey("dbo.Categories", Function(t) t.CategoryID, cascadeDelete := True) _
                .Index(Function(t) t.CategoryID)
            CreateTable(
                "dbo.Categories",
                Function(c) New With
                    {
                        .ID = c.Int(nullable := False, identity := True),
                        .Name = c.String()
                    }) _
                .PrimaryKey(Function(t) t.ID)
        End Sub

        Public Overrides Sub Down()
            DropForeignKey("dbo.Articles", "CategoryID", "dbo.Categories")
            DropIndex("dbo.Articles", New String() { "CategoryID" })
            DropTable("dbo.Categories")
            DropTable("dbo.Articles")
        End Sub
    End Class
End Namespace
```

Die meisten der Methoden, die hier zum Einsatz kommen, stammen aus dem Namespace **System.Data.Entity.Migrations**. Die Methode **Up** enthält zwei **CreateTable**-Anweisungen, mit denen die Tabellen **Articles** und **Categories** erstellt werden sollen. Wir haben also hier eine Möglichkeit, Tabellen und ihre Felder sowie die Restriktionen objektorientiert anzulegen statt mit T-SQL-Anweisungen. Letztendlich werden die Befehle natürlich auch wieder in T-SQL übersetzt, aber für manchen Entwickler, der sich vielleicht nicht mit T-SQL anfreunden möchte, bietet sich hier eine Alternative. Die Methode **Down** enthält vier Anweisungen, mit denen nacheinander die Restriktionen und die Tabellen gelöscht werden.

Änderungen in die Datenbank übertragen

Schließlich wollen Sie die Änderungen im Entity Data Model, die wir durch die **Add-Migration**-Anweisung in die Klasse mit den Methoden **Up** und **Down** geschrieben haben, noch in das Datenmodell der Datenbank übertragen. Dazu rufen wir die folgende Anweisung auf:

```
Update-Database
```

Nun werden die **Up**- und **Down**-Methoden aller seit dem letzten Aufruf von **Update-Database** angelegten Migrationsklassen ausgeführt. In unserem Fall wird die Datenbank nun überhaupt erst angelegt, denn bisher haben wir dies ja noch nicht erledigt. Auch **Update-Database** dauert einige Sekunden und schließlich erhalten Sie das Ergebnis des Aufrufs in der Paket-Manager-Konsole:

```
PM> update-database
Geben Sie das Flag '-Verbose' an, um die auf die Zieldatenbank angewendeten SQL-Anweisungen anzuzeigen.
Explizite Migrationen werden angewendet: [201811070831578_init].
Die explizite Migration wird angewendet: 201811070831578_init.
Die Seed-Methode wird ausgeführt.
```

Was ist nun geschehen? Dazu werfen wir einen Blick in den SQL Server-Objekt-Explorer (siehe Bild 4). Hier finden wir unter **Datenbanken** die neue Datenbank namens **DatabaseMigrations.ArticleContext**, die überraschenderweise nicht nur die zwei Tabellen **Categories** und **Articles** enthält, sondern noch eine weitere Tabelle namens **__MigrationHistory**. Das hört sich logisch an: Hier werden vermutlich Informationen über den aktuellen Stand der Datenbank gespeichert.

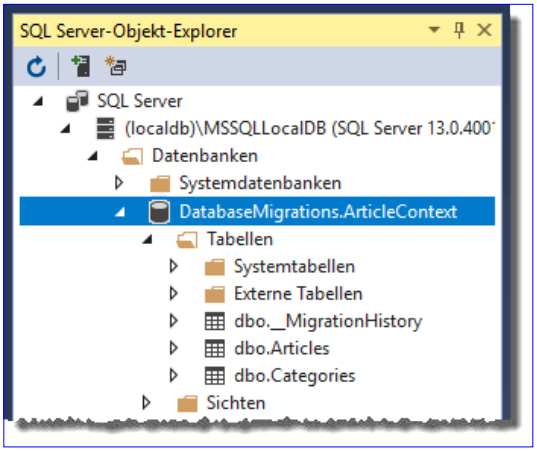


Bild 4: Die neue Datenbank im SQL Server-Objekt-Explorer

Schauen wir uns den Inhalt dieser Tabelle an (siehe Bild 5), finden wir genau einen Datensatz, der den Namen der bei **Add**-Migration erstellten Datei ohne Dateiendung enthält (**201811070831578_init**) sowie ein Feld namens **ContextKey**, der die **Configuration**-Klasse im Projekt angibt (**DatabaseMigrations.Migrations.Configuration**). Diese Tabelle bildet also die Grundlage, wenn es später darum geht, zu prüfen, welche der im Ordner **Migrations** angelegten Klassen bereits auf die Datenbank angewendet wurden und welche nicht. Die **Add-Migration**-Anweisung fügt die neue Klasse mit den **Up**- und **Down**-Methoden

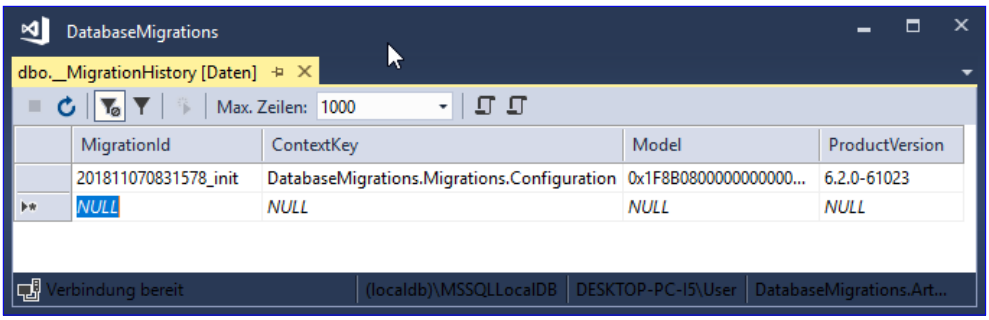


Bild 5: Datensatz für die erste Migration in der Tabelle **__MigrationHistory**

Von Access zu Entity Framework: Datenmodell

Viele Leser dieses Magazins programmieren auch mit Access. Der eine oder andere hat vielleicht sogar eigene Anwendungen oder Anwendungen von Kunden auf Access-Basis, die er gern in Form eines WPF- oder ASP.NET-Projekts umsetzen würde. Das Problem: Der Zugriff auf die Access-Datenbank ist unter .NET nur begrenzt möglich, die tolle Datenzugriffstechnologie Entity Framework beispielsweise unterstützt Access-Datenbanken nicht. Dafür unterstützt es allerdings SQL Server-Datenbanken. Wie gehen wir also vor? Wir migrieren die Access-Datenbanken zum SQL Server und bauen dann ein Entity Data Model auf Basis dieser Datenbank. Es geht allerdings auch anders: Sie könnten auch ein paar Routinen in VBA schreiben, die ein Entity Data Model direkt aus Access heraus auf Basis des gewünschten Datenmodells erzeugen. Dieser Artikel zeigt, wie letztere Möglichkeit funktioniert.

Als Beispiel habe ich eine kleine Bestellverwaltung zusammengestellt, welche die wesentlichen Tabellen enthält (siehe Bild 1).

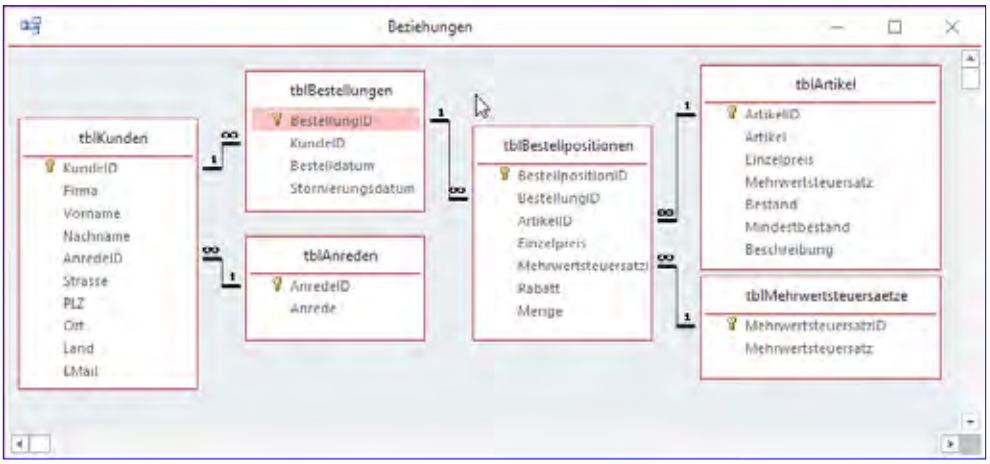


Bild 1: Beispiel für ein zu migrierendes Datenmodell

Ziel ist es, aus diesem Datenmodell und den enthaltenen Daten ein Entity Data Model mit je einer Klasse für jede Tabelle und einer Liste der benötigten **DbSet**-Auflistungen zu erstellen.

Außerdem wollen wir die Befehle für eine **Seed**-Methode zusammenstellen, die notwendig ist, um die Daten aus den Access-Tabellen dann beim Initialisieren des Entity Data Models in die zu erstellende Datenbank zu schreiben. Die Grundlagen zur Initialisierung einer Datenbank finden Sie im Artikel **Entity Framework: Datenbankinitialisierung**.

Was wollen wir also genau tun? Wir möchten beispielsweise für die Tabelle **tblAnreden**, deren Entwurf Sie in Bild 2 sehen, zwei Dinge erzeugen:

- die Definition einer Klasse und
- die Definition eines **DbSet**-Objekts.

Die Klassendefinition soll beispielsweise wie folgt aussehen:

```
Public Class Anrede
    Public Property AnredeID As System.Int32
    Public Property Anrede As System.String
End Class
```

Die Definition des **DbSet**-Objekts lautet so:

```
Public Overridable Property Anreden() As
DbSet(Of Anrede)
```

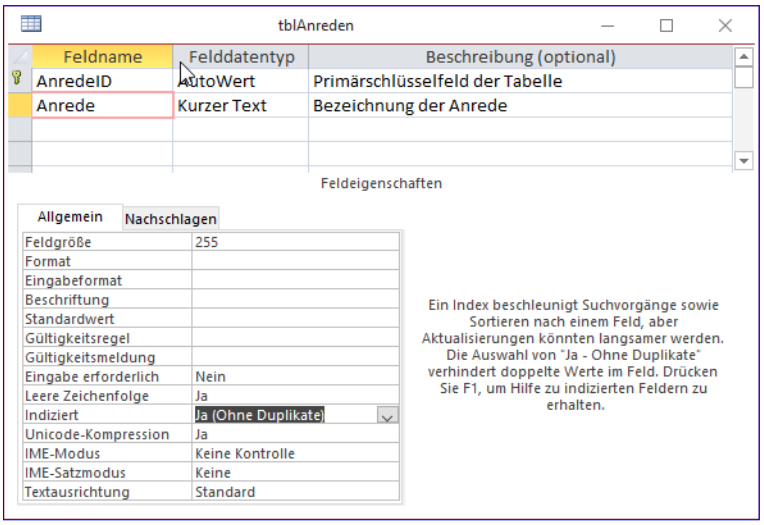


Bild 2: Zu migrierende Tabelle

Hier würden wir von einer automatischen Erfassung des Tabellennamens, der Felder und des Primärschlüssels ausgehen und von ein paar Anpassungen, um die Migration einigermaßen tauglich zu gestalten und beispielsweise nicht das Präfix **tbl** mitzuschleppen. Wir sehen hier, dass die Klasse beispielsweise **Anrede** heißt. Diese Bezeichnung können wir nur schwer aus dem Tabellennamen extrahieren, also holen wir ihn aus dem Namen des Primärschlüsselfeldes (hier **AnredeID**), von dem wir lediglich den Zusatz **ID** entfernen. Die beiden Felder belassen wir im ersten Schritt bei **AnredeID** und **Anrede**. Optimaler wäre beispielsweise **ID** und **Name**. Ersteres, weil wir in der objektorientierten Programmierung etwa über **<Objektname>.<Feldname>** auf die Eigenschaften einer Entität zugreifen und **Anrede.AnredeID** schlicht redundante Daten enthält – **Anrede.ID** wäre viel schöner. Und **Anrede.Anrede** ist ebenfalls optimierungsfähig. Für die Deklaration des **DbSet**-Objekts verwenden wir aktuell den Namen der Tabelle ohne das Präfix **tbl** und für den Typ der im **DbSet** enthaltenen Daten verwenden wir wieder den aus dem Primärschlüsselfeld ermittelten Namen, also **Anrede**. Gerade bei den Bezeichnern für die DbSets und die Entitäten muss man darauf achten, ob sich Plural und Singular unterscheiden. Das ist beispielsweise bei **Artikel** schwierig (ein Artikel/zwei Artikel), weshalb man, wenn man schon weiß, dass man mal objektorientiert mit dem Datenmodell seiner Datenbank arbeiten möchte, besser gleich eine Bezeichnung wie **Produkt**/**Produkte** verwendet. In unserer Beispieldatenbank verwenden wir aber weiter **tblArtikel**, gerade weil wir auch zeigen wollen, wie Sie solche Probleme durch Umbenennungen lösen können.

Beispieldatenbank

Die Beispieldatenbank enthält die im Beziehungen-Fenster von Access abgebildeten Tabellen und Beziehungen. Diese wollen wir nun, möglichst ohne manuellen Eingriff, in die entsprechenden Entitäten und DbSets umwandeln. Dazu können wir zwei Wege gehen: Entweder wir greifen von einem Visual Studio-Projekt aus auf die Datenbank zu oder wir bauen unsere Routinen direkt in der Access-Datenbank. Mir selbst geht VBA beim Zugriff auf Tabellen, Felder, Beziehungen und so weiter immer noch viel schneller von der Hand als mit VB oder C#. Außerdem wäre der Zugriff, wie er für das Auslesen der Access-Tabellen nötig wäre, eine Technik, die wir prinzipiell unter .NET nicht mehr benötigen – hier wollen wir Datenbanksysteme wie SQL Server oder SQLite nutzen, auf die wir mit dem Entity Framework zugreifen können.

Außerdem dürfte das Programmieren von Routinen, die uns aus dem Access-Datenmodell ein Entity Data Model macht, mit einer Menge Testen und Ändern verbunden sein. Das geht unter Access wesentlich schneller und komfortabler als in Visual Studio. In Access schreiben wir einfach die Prozeduren und klicken auf **F5**, um diese auszuführen, in Visual Studio müssten wir

immer das Projekt neu starten, die Ausführung prüfen, das Projekt wieder beenden, den Code anpassen und das Ganze immer wieder von vorn beginnen. Also entscheiden wir uns in diesem Fall für die ältere, aber pragmatischere Vorgehensweise.

.NET-Projekt zum Testen vorbereiten

Damit wir die gleich unter Access generierten Klassen im .NET-Projekt auf ihre Tauglichkeit prüfen können, legen wir gleich ein neues Projekt des Typs **VB|WPF-App** mit dem Namen **AccessZuEF** an. Diesem fügen wir ein neues Objekt des Typs **ADO.NET Entity Data Model** namens **BestellverwaltungContext** mit dem Typ **Leeres Code First-Modell** hinzu. Dies legt automatisch die Klasse **BestellverwaltungContext.vb** für uns an, der wir dann gleich unsere Liste der **DbSet**-Elemente hinzufügen können. Der Einfachheit halber packen wir unsere **Class**-Elemente zum Testen auch erst einmal hier herein.

Zugriff auf die Tabellen, Felder und Beziehungen

Den Zugriff auf die für uns interessanten Elemente gestalten wir mit der DAO-Bibliothek und VBA. Wir arbeiten uns nun Schritt für Schritt an die Umwandlung des Datenmodells und der enthaltenen Daten an die entsprechenden .NET-Klassen heran. Zuerst sehen wir uns ein paar Hilfsfunktionen an, die uns auf dem Weg behilflich sind.

Datentypen umwandeln

Unter dem Link **https://support.microsoft.com/de-de/help/320435/info-oledbtype-enumeration-vs-microsoft-access-data-types** finden wir eine Liste der Access-Datentypen und der entsprechenden Datentypen unter .NET. Daraus bauen wir uns eine kleine Funktion, die den Access-Datentyp als Parameter erwartet und den .NET-Datentyp als Ergebnis zurückliefert. Diese Funktion sieht wie folgt aus:

```
Public Function DataTypeNET(intDatatype As DataTypeEnum, strDatatype As String, bolArray As Boolean) As Boolean
    Dim bolDatatype As Boolean
    bolDatatype = True
    bolArray = False
    Select Case intDatatype
        Case dbText: strDatatype = "System.String"
        Case dbMemo: strDatatype = "System.String"
        Case dbByte: strDatatype = "System.Byte"
        Case dbBoolean: strDatatype = "System.Boolean"
        Case dbDate: strDatatype = "System.DateTime"
        Case dbCurrency: strDatatype = "System.Decimal"
        Case dbDecimal: strDatatype = "System.Decimal"
        Case dbDouble: strDatatype = "System.Double"
        Case dbGUID: strDatatype = "System.Guid"
        Case dbLong: strDatatype = "System.Int32"
        Case dbLongBinary
            strDatatype = "System.Byte"
            bolArray = True
        Case dbSingle: strDatatype = "System.Single"
        Case dbInteger: strDatatype = "System.Int16"
```

```
        Case dbBinary
            strDatatype = "System.Byte"
            bolArray = True
        Case Else
            bolDatatype = False
    End Select
    DataTypeNET = bolDatatype
End Function
```

Die Funktion erwartet drei Parameter, von denen nur der erste Parameter Daten an die Funktion schickt – die übrigen beiden liefern die Rückgabewerte. **intDataType** erwartet die Konstante beziehungsweise den entsprechenden Zahlenwert, den wir aus der Eigenschaft **Type** eines **Field**-Elements erhalten. **strDatatype** liefert den .NET-Datentyp als String zurück. Manche Datentypen werden in .NET als Byte-Array gestaltet. Wenn der Datentyp ein Array erfordert, markieren wir den dritten Parameter **bolArray** als **True**. Es kann auch sein, dass der Datentyp nicht vorhanden ist, was schlicht daran liegt, dass er in dieser Funktion nicht berücksichtigt wird – in diesem Fall kommt der eigentliche Rückgabewert der Funktion zum Einsatz, der den Datentyp **Boolean** hat.

Die Funktion stellt die Variable **bolDatatype**, die festlegt, ob die Funktion einen passenden Datentyp in .NET gefunden hat, standardmäßig auf **True** ein. Diese wird erst auf **False** eingestellt, wenn der Datentyp in der folgenden **Select Case**-Bedingung nicht gefunden wurde. Die Variable **bolArray** wird zuerst auf **False** eingestellt und erhält erst den Wert **True**, wenn der Datentyp ein Array benötigt. Das ist in diesem Fall nötig, da **bolArray** ja auch in der aufrufenden Prozedur verwendet wird und gegebenenfalls vom vorherigen Aufruf noch den Wert **True** enthält. Davon ab vergleicht die **Select Case**-Bedingung den mit **intDatatype** übergebenen Wert mit Konstanten wie **dbText**, **dbMemo**, **dbByte** und so weiter und trägt, wenn sie einen passenen Eintrag gefunden hat, den entsprechenden .NET-Datentyp in die Variable **strDatatype** ein. Am Ende wird schließlich der Wert von **bolDatatype** noch als Rückgabewert der Funktion festgelegt.

Primärschlüsselfeld ermitteln

Wir gehen davon aus, dass die Primärschlüsselfelder der Tabellen mit einer Bezeichnung versehen sind, die aus dem Singular der Bezeichnung besteht, die sich im Tabellennamen befindet, und der angehängten Zeichenkette **ID** – bei **tblAnreden** also etwa **AnredeID** (das ist zumindest die Konvention, die ich seit 20 Jahren in Magazin **Access im Unternehmen** und in all meinen anderen Veröffentlichungen über Access verwende). Um den Namen herauszubekommen, müssen wir nun allerdings noch zuverlässig bestimmen können, welches Feld das Primärschlüsselfeld ist. Wir gehen an dieser Stelle davon aus, dass jede Tabelle auch nur einen aus einem Feld bestehenden Primärschlüssel verwendet.

Die folgende Funktion namens **GetPrimaryKeys** ermittelt dennoch die Namen aller Primärschlüsselfelder einer Tabelle. Dazu erhält sie mit **strTable** den Tabellennamen als Parameter sowie **strPKs** als Rückgabeparameter. Die Funktion lädt ein **TableDef**-Objekt auf Basis des Namens der Tabelle und durchläuft dann alle **Index**-Elemente dieser Tabelle in einer **For Each**-Schleife. Wenn es sich bei dem Index um einen Primärschlüssel handelt (**Primary = True**), durchläuft die Funktion alle Felder des Indexes und fügt die Namen der Felder in der Variablen **strPKTemp** durch Semikola getrennt aneinander. Das Ergebnis wird mit dem Parameter **strPKs** zurückgeschickt. Der Rückgabewert der Funktion **GetPrimaryKeys** wird mit dem Wert **True** gefüllt, wenn es mindestens ein Primärschlüsselfeld gibt.

```
Public Function GetPrimaryKeys(strTable As String, strPKs As String) As Boolean
    Dim db As DAO.Database
    Dim tdf As DAO.TableDef
    Dim idx As DAO.Index
    Dim strPKTemp As String
    Dim i As Integer
    Set db = CurrentDb
    Set tdf = db.TableDefs(strTable)
    For Each idx In tdf.Indexes
        If idx.Primary Then
            For i = 0 To idx.Fields.Count - 1
                strPKTemp = strPKTemp & idx.Fields(0).Name & ";"
            Next i
        End If
    Next idx
    If Len(strPKTemp) = 0 Then
       Exit Function
    End If
    strPKs = Left(strPKTemp, Len(strPKTemp) - 1)
    GetPrimaryKeys = True
End Function
```

Die folgende Funktion **GetPrimaryKey** unterscheidet sich im Namen nur durch den Singular gegenüber dem Plural von der Funktion **GetPrimaryKey**. Was macht diese Funktion? Sie ruft in erster Linie die zuvor beschriebene Funktion **GetPrimaryKeys** auf und prüft, ob die mit dem Parameter **strTable** übergebene Tabelle nur um einen oder mehrere Primärschlüssel enthält. Falls es mehrere sind, wird mit **strPK** eine leere Zeichenkette und mit dem Rückgabewert der Wert **False** zurückgegeben. Falls es sich um ein einziges Primärschlüsselfeld handelt, wird mit **strPK** der Name des Primärschlüsselfeldes sowie mit dem Rückgabewert der Wert **True** zurückgeliefert:

```
Public Function GetPrimaryKey(strTable As String, strPK As String) As Boolean
    Dim strPKs As String
    If GetPrimaryKeys(strTable, strPKs) = True Then
        If InStr(1, strPKs, ";") Then
            Exit Function
        Else
            strPK = strPKs
            GetPrimaryKey = True
        End If
    Else
        Exit Function
    End If
End Function
```

In der aufrufenden Prozedur, die wir uns gleich im Anschluss ansehen, können wir dann aufgrund der Rückgabewerte entweder die Definition der Klasse und der **DbSet**-Anweisung zusammenstellen oder eine Meldung ausgeben, dass die Bedingungen für das Erstellen der Elemente wegen keinem Primärschlüsselfeld oder mehr als einem Primärschlüsselfeld nicht gegeben sind.

Klassen und DbSet-Deklarationen zusammenstellen

Die folgende Prozedur erledigt den eigentlichen Teil der Arbeit: Sie nutzt die Ergebnisse der zuvor beschriebenen Funktionen und weitere Daten, um die Klassendefinitionen und die **DbSet**-Deklarationen zusammenzustellen. Im ersten Schritt wollen wir diese noch mit **Debug.Print** im Direktbereich ausgeben. Später werden wir direkt entsprechende Dateien daraus erzeugen, die Sie nur noch aus dem Windows-Explorer in den Projektmappen-Explorer des Ziel-Projekts in Visual Studio ziehen. Die Prozedur heißt **EDMErstellen** und durchläuft nach dem Deklarationsteil zunächst einmal alle **TableDef**-Elemente, also alle Tabellendefinitionen, der mit **CurrentDb** referenzierten Datenbank – also der aktuell geöffneten Datenbank:

```
Public Sub EDMErstellen()
    Dim db As DAO.Database
    Dim tdf As DAO.TableDef
    Dim fld As DAO.Field
    Dim strDbSets As String
    Dim strEntity As String
    Dim strPK As String
    Dim strDatatype As String
    Dim bolArray As Boolean
    Dim strEntities As String
    Dim bolDatatypeExists As Boolean
    Dim strArray As String
    Set db = CurrentDb
    For Each tdf In db.TableDefs
```

Hier prüft die Prozedur dann, ob der Name der Tabelle nicht mit **MSys** beginnt, denn das sind die Systemtabellen, die wir nicht berücksichtigen wollen. Beginnt die Tabelle nicht mit **MSys**, prüfen wir, ob die Tabelle genau ein Primärschlüsselfeld enthält. Falls nicht, geben wir im Direktbereich einen entsprechenden Hinweis aus, den wir als Kommentar kennzeichnen:

```
        If Not Left(tdf.Name, 4) = "MSys" Then
            strPK = ""
            If Not GetPrimaryKey(tdf.Name, strPK) Then
                Debug.Print "    '****"
                Debug.Print "    'Mehrere PKs in " & tdf.Name & ". Die Klasse wird nicht erstellt."
```

Anderenfalls, also wenn genau ein Primärschlüsselfeld vorhanden ist, ermitteln wir den Namen der Entität aus dem Namen des Primärschlüsselfelds, von dem wir zuvor die Zeichenkette **ID** entfernen (also etwa **Anrede** statt **AnredeID**). Den Namen für die **DbSet**-Auflistung entnehmen wir dem Tabellennamen und entfernen dort das Präfix **tbl** (also **Anreden** statt **tblAnreden**). Die Zeile für das **DbSet** können wir damit schon zusammenstellen. Wir sammeln diese Zeilen in der Variablen **strDbSets**:

```
        Else
            strEntity = Replace(strPK, "ID", "")
            strEntities = Replace(tdf.Name, "tbl", "")
            strDbSets = strDbSets & "Public Overridable Property " & tdf.Name & "() As DbSet(Of " & strEntity & ")" _
                & vbCrLf
```

Danach starten wir mit der Ausgabe der Klassendefinition und beginnen mit Public Class und dem Namen der Entität:

```
            Debug.Print "Public Class " & strEntity
```

Danach durchlaufen wir in einer **For Each**-Schleife alle Felder der aktuell in **tdf** befindlichen Tabelle und ermitteln mit der Funktion **DataTypeNET** den Datentyp für .NET für das aktuelle Feld. Wenn **bolArray** den Wert **True** hat, tragen wir in die Variable **strArray** die Zeichenkette **()** ein, sonst eine leere Zeichenkette. Dann prüfen wir, ob **bolDataTypeExists** den Wert **True** hat und geben die Zeile für die Deklaration der Eigenschaft dieses Feldes im Direktbereich aus.

Falls nicht, tragen wir hier einen Text ein, der darauf hinweist, dass kein Datentyp für dieses Feld gefunden werden konnte:

```
            For Each fld In tdf.Fields
                bolDatatypeExists = DataTypeNET(fld.Type, strDatatype, bolArray)
                strArray = IIf(bolArray, "()", "")
                If bolDatatypeExists Then
                    Debug.Print "    Public Property " & fld.Name & strArray & " As " & strDatatype
                Else
                    Debug.Print "    'Kein Datentyp für " & fld.Name & " gefunden."
                End If
            Next fld
```

Nachdem wir alle Felder der aktuellen Tabelle durchlaufen haben, schließen wir die Definition der Klasse mit **End Class** im Direktbereich ab und fahren mit der nächsten Tabelle fort. Anschließend geben wir noch die in **strDbSets** gesammelten Deklarationen der DbSets aus:

```
            Debug.Print "End Class"
        End If
    End If
Next tdf
Debug.Print strDbSets
End Sub
```

Wenn wir die Prozedur nun aufrufen, erhalten wir im Direktbereich schon recht ansehnliche Ergebnisse, die wir so auch einmal in die Kontext-Klasse in unserem dafür vorbereiteten .NET-Projekt einfügen können – allein um zu prüfen, ob die enthaltenen Informationen schon einmal die korrekte Syntax aufweisen.

Das Ergebnis sieht dann beispielsweise wie in Bild 3 aus. Zumindest werden keine Fehlermarkierungen angezeigt – das ist schon einmal positiv!

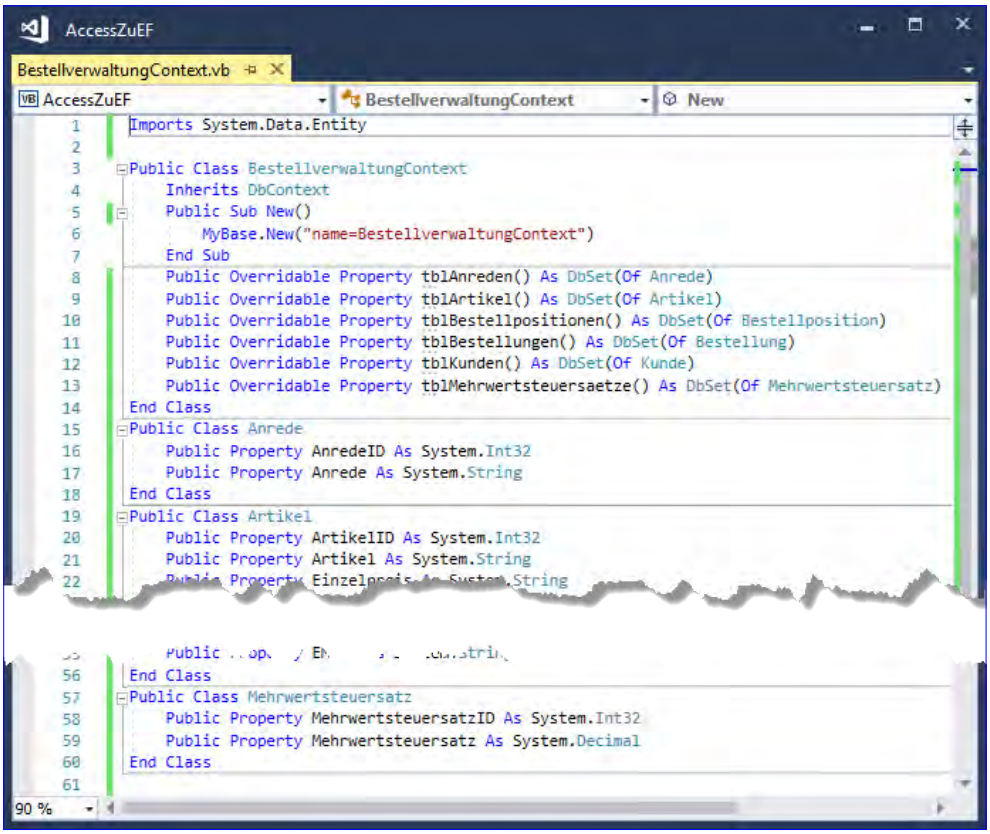


Bild 3: Erster Test mit den Definitionen im .NET-Projekt

Code First-Datenbank intialisieren

An dieser Stelle können wir, auch wenn natürlich noch einige Informationen fehlen, ja auch durchaus schon einmal probieren, die Code First-Datenbank zu initialisieren. Wie das im Detail gelingt, erfahren Sie im Artikel **Entity Framework: Datenbankinitialisierung**. Wir machen die Kurzfassung: Sie blenden die Paket-Manager-Konsole in Visual Studio ein und geben zunächst diesen Befehl ein:

```
Enable-Migrations
```

Das Ergebnis ist etwas ernüchternd und liefert uns noch einige Zusatzarbeit:

Während der Modellgenerierung wurde mindestens ein Überprüfungsfehler erkannt:

```
Anrede: Name: Der Name 'Anrede' kann nicht in Typ 'AccessZuEF.Anrede' verwendet werden. Elementnamen dürfen nicht mit dem einschließenden Typ übereinstimmen.
Artikel: Name: Der Name 'Artikel' kann nicht in Typ 'AccessZuEF.Artikel' verwendet werden. Elementnamen dürfen nicht mit dem einschließenden Typ übereinstimmen.
Mehrwertsteuersatz: Name: Der Name 'Mehrwertsteuersatz' kann nicht in Typ 'AccessZuEF.Mehrwertsteuersatz' verwendet werden. Elementnamen dürfen nicht mit dem einschließenden Typ übereinstimmen.
```

Wir können also keine Eigenschaftsnamen verwenden, die genauso lauten wie das Element selbst. Wenn also die Entität **Anrede** heißt, darf die Entität keine Eigenschaft namens **Anrede** enthalten. Wie können wir das im Code berücksichtigen? Wir können hier schon eine der weiter oben als Optimierung betrachtete Änderung vornehmen. Genau genommen hat die Meldung uns auch einen Hinweis geliefert, wie wir herausfinden, welches der Felder wir mit der Bezeichnung **Name** versehen können

Von Access zu Entity Framework: Daten

Viele Leser dieses Magazins programmieren auch mit Access. Daher haben wir im Artikel »Von Access zu Entity Framework: Datenmodell« bereits gezeigt, wie Sie die meisten Elemente eines Datenmodells in Klassen für ein Entity Data Model überführen, die Sie dann wiederum zum Erstellen einer SQL Server-Datenbank per Code First nutzen können. Was fehlt, sind allerdings noch die Daten in diesen Tabellen. Wie Sie den Code erstellen, um auch die Daten über eine entsprechend Seed-Methode in die Datenbank zu schreiben, erfahren Sie in diesem Artikel.

Im Artikel **Von Access zu Entity Framework: Datenmodell** haben wir uns darum gekümmert, das Datenmodell unserer kleinen Access-Beispielanwendung aus Bild 1 zuerst in die Klassen eines Entity Data Models zu überführen. Dann haben wir mit den Methoden **Add-Migration** und **Update-Database** in der Paket-Manager-Konsole dafür gesorgt, dass diese Klassen zusammen mit den passenden **DbSet**-Auflistungen in eine neue SQL Server-Datenbank überführt wurden. Die dort vorgestellten Routinen sind sicher noch nicht perfekt, aber sie sind eine gute Grundlage, die Sie selbst erweitern können – oder Sie teilen uns einfach mit, bei welchen Eigenarten von Tabellendefinitionen oder Datenmodellen die Erstellung des Entity Data Models oder der SQL Server-Datenbank noch nicht funktioniert.

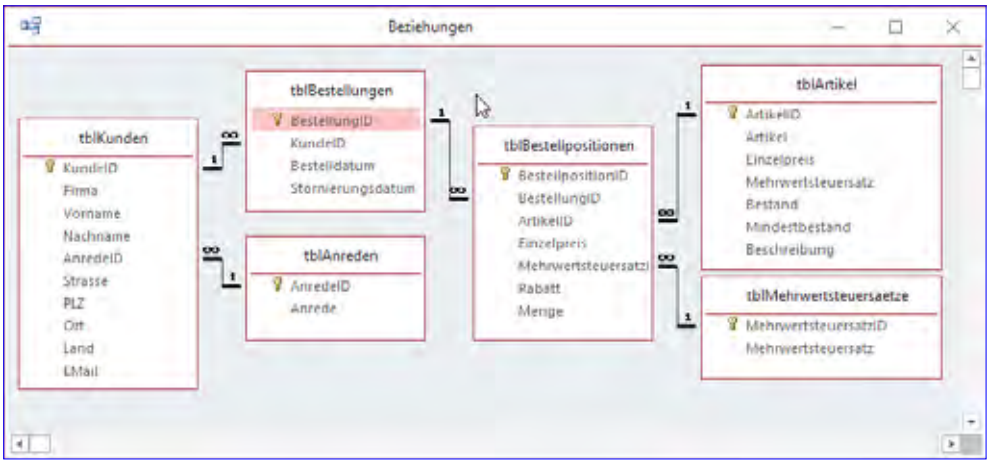


Bild 1: Beispiel für ein zu migrierendes Datenmodell

Im vorliegenden Artikel nun wollen wir die Daten in diesen Tabellen in eine Form bringen, in der wie diese beim Erstellen der Datenbank von Visual Studio aus mit der **Seed**-Methode direkt in die frisch angelegten Tabellen schreiben können. Die Grundlagen dafür erhalten Sie im Artikel **Entity Framework: Tabellen füllen** sowie in den beiden Artikeln **Entity Framework: Datenbankinitialisierung** und **Entity Framework: Datenbankmigration**.

Anlegen der Daten für eine einzelne Tabelle ohne Fremdschlüsselfelder

Wir wollen uns langsam an die Lösung der Aufgabe heranbewegen und uns zuerst einmal eine einfache Tabelle heraussuchen, deren Daten wir dann über den Weg der **Seed**-Methode etwa der **Configuration**-Klasse im **Migrations**-Ordner nach dem Erstellen des Datenmodells in die Datenbank schreiben (wie Sie diesen Ordner erstellen, erfahren Sie im Artikel **Entity Framework: Datenbankmigration**).

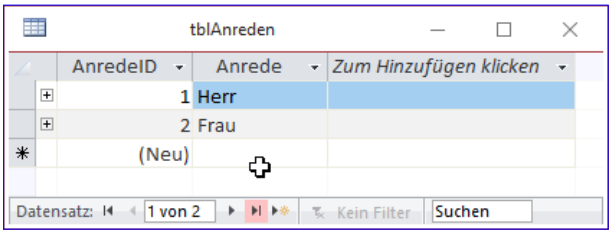


Bild 2: Die Tabelle **tblAnreden** der Ausgangsdatenbank

Die Tabelle **tblAnreden** unserer Ausgangsdatenbank enthält nur zwei Datensätze (siehe Bild 2). Diese wollen wir nun in einem ersten Schritt in Anweisungen schreiben, die wir in die **Seed**-Methode der .NET-Anwendung schreiben können. Wir schauen uns erst einmal an, wie die fertige **Seed**-Methode anschließend aussehen soll:

```
Namespace Migrations
    Friend NotInheritable Class Configuration
        Inherits DbMigrationsConfiguration(Of BestellverwaltungContext)
        Public Sub New()
            AutomaticMigrationsEnabled = False
        End Sub
        Protected Overrides Sub Seed(context As BestellverwaltungContext)
            context.Anreden.AddOrUpdate(Function(x) x.ID,
                New Anrede() With {.Name = "Herr"},
                New Anrede() With {.Name = "Frau"}
            )
        End Sub
    End Class
End Namespace
```

Wir sollten uns überlegen, ob wir per VBA immer die komplette Klasse namens **Configuration** wie oben abgebildet erstellen sollen oder nur die **Seed**-Methode in dieser Klasse. Praktischer ist es vermutlich, gleich die komplette Klasse zu erstellen, damit man entweder – im ersten Schritt – die Ausgabe dieser Klasse aus dem Direktbereich des VBA-Editors von Access komplett kopieren und als Ganzes in diese Klasse einsetzen kann. Im zweiten Schritt könnten wir den Code direkt in die entsprechende Datei schreiben, die dann von Visual Studio aktualisiert eingelesen wird.

Die **Seed**-Methode enthält hier eine Anweisung, bei der wir die **AddOrUpdate**-Methode der Auflistung **Anreden** aufrufen und dieser eine Funktion übergeben, welche jeweils ein neues Objekt auf Basis der Klasse **Anrede** enthält. Für diese legen wir dann jeweils die Anrede fest.

Wir bauen zunächst eine Prozedur, welche den Rahmen im Direktfenster ausgibt, also die **Imports**-Anweisungen, den Namespace, die Klasse und die enthaltenen Methoden – mit Ausnahme der **Seed**-Methode:

```
Public Sub SeedErstellen()
    Dim strSeed As String
    strSeed = strSeed & "Imports System" & vbCrLf
    strSeed = strSeed & "Imports System.Data.Entity"
    strSeed = strSeed & "Imports System.Data.Entity.Migrations"
    strSeed = strSeed & "Imports System.Linq"
    strSeed = strSeed & ""
    strSeed = strSeed & "Namespace Migrations"
    strSeed = strSeed & "    Friend NotInheritable Class Configuration"
```

```
    strSeed = strSeed & "        Inherits DbMigrationsConfiguration(Of BestellverwaltungContext)"
    strSeed = strSeed & "        Public Sub New()"
    strSeed = strSeed & "            AutomaticMigrationsEnabled = False"
    strSeed = strSeed & "        End Sub"
    strSeed = strSeed & "        Protected Overrides Sub Seed(context As BestellverwaltungContext)"
    strSeed = strSeed & SeedData
    strSeed = strSeed & "        End Sub"
    strSeed = strSeed & "    End Class"
    strSeed = strSeed & "End Namespace"
End Sub
```

Das ist noch kein Hexenwerk. Interessant ist die hier eingebettete Prozedur **SeedData**. Diese stellt die Anweisungen zusammen, die wir zur Methode **Seed** hinzufügen wollen. Statisch sieht das für unsere Tabelle **tblAnreden** zunächst wie folgt aus:

```
Public Sub SeedData()
    Dim strSeedData As String
    strSeedData = strSeedData & "            context.Anreden.AddOrUpdate(Function(x) x.ID,"
    strSeedData = strSeedData & "                New Anrede() With {.Name = ""Herr""},"
    strSeedData = strSeedData & "                New Anrede() With {.Name = ""Frau""}"
    strSeedData = strSeedData & "            )"
    SeedData = strSeedData
End Sub
```

Um dies aus VBA heraus dynamisch für die Tabelle **tblAnreden** zu erzeugen, müssen wir dies allerdings mit einigen erst zur Laufzeit ermittelten Daten füllen. Das erledigen wir in den folgenden Schritten.

Seed für mehrere Tabellen

Die Prozedur **SeedData** wandeln wir in eine Funktion um, die das Ergebnis an die aufrufende Prozedur zurückgeben soll. Sie enthält dafür für jede Tabelle der Datenbank einen Aufruf der Tabelle **SeedData** und fügt das Ergebnis jeweils an die Zeichenkette **strSeed** an. Die Aufrufe der Tabellen erledigen wir in einer **For Each**-Schleife über alle **TableDef**-Elemente der mit **CurrentDb** referenzierten aktuellen Datenbank. In der Schleife prüfen wir zunächst, ob es sich bei der aktuellen Tabelle nicht um eine Systemtabelle von Access handelt. Nur wenn das nicht der Fall ist, rufen wir die Funktion **SeedData** mit dem Namen der Tabelle auf. Den Namen der Tabelle ermitteln wir dabei mit der Eigenschaft **Name** des **TableDef**-Objekts.

```
Public Sub SeedErstellen()
    ...
    strSeed = strSeed & "        Protected Overrides Sub Seed(context As BestellverwaltungContext)" & vbCrLf
    For Each tdf In db.TableDefs
        If Not Left(tdf.Name, 4) = "MSYS" Then
            strSeed = strSeed & SeedData(tdf.Name) & vbCrLf
        End If
```

```
    Next tdf    strSeed = strSeed & "        End Sub" & vbCrLf
    strSeed = strSeed & "    End Class" & vbCrLf
    strSeed = strSeed & "End Namespace"
    Inzwischenablage strSeed
End Sub
```

Am Ende der Prozedur nutzen wir die Prozedur **InZwischenablage**, um den Inhalt der Variablen **strSeed**, in der wir die Anweisungen für die **Configuration.vb**-Klasse gesammelt haben, in die Zwischenablage zu kopieren. Das ist eine bessere Lösung als die Ausgabe im Direktbereich, weil dieser nur eine begrenzte Anzahl Zeichen aufnehmen kann und bei vielen Daten der obere Teil der Ausgabe dann nicht mehr verfügbar ist. Die Prozedur **InZwischenablage** finden Sie im Modul **mdlZwischenablage**.

Die Funktion SeedData zum Zusammenstellen der AddOrUpdate-Aufrufe

Die Funktion **SeedData** bohren wir gegenüber der vorherigen Version erheblich auf. Als Erstes fällt der Parameter **strTabelle** auf, welcher die Information enthält, für die Datensätze welcher Tabelle die **AddOrUpdate**-Methoden zusammengestellt werden sollen.

```
Public Function SeedData(strTabelle As String) As String
    Dim db As DAO.Database
    Dim rst As DAO.Recordset
    Dim strSeedData As String
    Dim strPK As String
    Dim strFelder As String
    Dim fld As DAO.Field
    Dim strEntity As String
    Dim strEntities As String
    Dim strProperty As String
```

Nach der Deklaration der Variablen speichern wir einen Verweis auf die aktuelle Datenbank in der Variablen **db**. Dann prüfen wir mit der Funktion **GetPrimaryKey**, die wir bereits im Artikel **Von Access zu Entity Framework: Datenmodell** erläutert haben, ob die gerade untersuchte Tabelle genau ein Primärschlüsselfeld bereitstellt. Ist das der Fall, wird eine entsprechende Meldung im Direktbereich ausgeben und die Funktion wird mit **Exit Function** verlassen:

```
    Set db = CurrentDb
    If Not GetPrimaryKey(strTabelle, strPK) Then
        Debug.Print "    'Mehrere PKs in " & strTabelle & ". Die Klasse wird nicht erstellt."
        Exit Function
```

Kann jedoch genau ein Primärschlüsselfeld gefunden werden, landet dies in der Variablen **strPK**. Daraus ermitteln wir zunächst den Namen der Entität, indem wir hinten die Zeichenkette **ID** entfernen. Außerdem ermittelten wir den Namen der Tabelle, indem wir vom Tabellenname vorn **tbl** abtrennen und das Ergebnis in der Variablen **strEntitaeten** speichern:

```
Else
    strEntity = Replace(strPK, "ID", "")
    strEntities = Replace(strTabelle, "tbl", "")
```

Damit starten wir in die Ermittlung der benötigten Daten, indem wir ein Recordset auf Basis der mit **strTabelle** übergebenen Tabelle erstellen. Außerdem tragen wir die erste Zeile in die Variable **strSeedData** ein, welche zum Beispiel bei der Tabelle **tblAnreden** nun **context.Anreden.AddOrUpdate(Function(x) x.ID,** lautet:

```
Set rst = db.OpenRecordset("SELECT * FROM " & strTabelle, dbOpenDynaset)
strSeedData = strSeedData & "            context." & strEntities & ".AddOrUpdate(Function(x) x.ID," & vbCrLf
```

Danach durchlaufen wir in einer **Do While**-Schleife alle Datensätze der aktuell bearbeiteten Tabelle. Im ersten Schritt legen wir die Zeile **New Anrede() With {** an:

```
Do While Not rst.EOF
    strSeedData = strSeedData & "                New " & strEntity & "() With {"
```

Dann leeren wir die Variable **strFelder**, die wir anschließend mit den Zuweisungen für die einzelnen Felder mit den Werten des Recordsets füllen:

```
    strFelder = ""
```

In einer **For Each**-Schleife über alle Felder des Recordsets durchlaufen wir dann ein Feld nach dem anderen. Dabei prüfen wir zuerst, ob es sich beim aktuellen Feld nicht um das Primärschlüsselfeld handelt. Falls nicht, prüfen wir dann, ob der Feldname dem Wert aus **strEntity** entspricht. Das ist zum Beispiel beim Feld **Anrede** der Tabelle **tblAnreden** der Fall. Das würde bedeuten, dass das Feld genauso heißen würde wie die Entität, also **Anrede**. Wie wir spätestens im Artikel **Von Access zu Entity Framework: Datenmodell** gelernt haben, ist das allerdings per Definition nicht erlaubt. Für diesen Fall haben wir in dem Artikel vorgesehen, dass wir das Feld dann einfach **Name** nennen (man hätte auch **Bezeichnung** nehmen können). Um dem gerecht zu werden, verwenden wir hier die Variable **strProperty**, die wir dann entweder mit dem tatsächlichen Feldnamen füllen oder, wenn der Feldname dem Entitätsnamen entspricht, einfach mit **Name**:

```
    For Each fld In rst.Fields
        If Not fld.Name = strPK Then
            If Not fld.Name = strEntity Then
                strProperty = fld.Name
            Else
                strProperty = "Name"
            End If
```

Danach machen unterscheiden wir in einer **Select Case**-Bedingung nach dem Datentyp der Felder. Hier benötigen wir beispielsweise einen **Case**-Zweig für Zahlen-Datentypen, einen für Text-Datentypen, einen für das Datum und einen für Währun-

Von Access zu EF: Step by step

Im Artikel »Von Access zu Entity Framework: Datenmodell« zeigen wir, wie Sie von Access aus auf Basis des aktuellen Datenmodells Klassen für ein Entity Data Model erstellen. In »Von Access zu Entity Framework: Daten« zeigen wir, wie Sie noch eine Seed-Anweisung hinzufügen, welche die Daten der Access-Datenbank in die auf Basis des Entity Data Models erstellte Datenbank füllt. Im vorliegenden Artikel nun liefern wir nochmal eine Kurzanleitung, wie Sie die in den Artikeln vorgestellten Techniken Schritt für Schritt nutzen, um ein .NET-Projekt mit Entity Data Model samt Daten aus der Access-Datenbank zu erstellen.

Um aus einer Access-Datenbank wie in Bild 1 ein Entity Data Model zu machen, daraus eine SQL Server-Datenbank zu generieren und dies auch noch mit den Daten der Access-Datenbank zu füllen ist gar nicht so schwer, wenn Sie die Lösungen aus den oben genannten Artikeln verwenden.

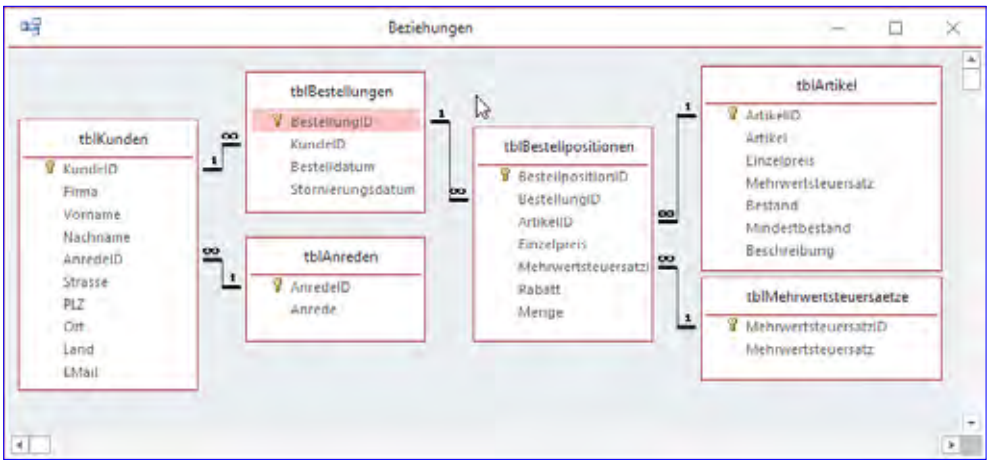


Bild 1: Zu migrierende Access-Datenbank

Projekt erstellen

Als Erstes erstellen wir ein Projekt auf Basis der Vorlage **Visual Basic|Windows Desktop|WPF-App** und nennen es beispielsweise **AccessZuEF**. Dann fügen wir diesem über den Kontextmenü-Eintrag **Hinzufügen|Neues Element...** des Projekt-Elements im Projektmappen-Explorer ein neues Element des Typs **ADO.NET Entity Data Model** hinzu, etwa namens **Bestellverwaltung**Context. Im **Assistent für Entity Data Model** wählen wir für das Modell den Typ **Leeres Code First-Modell** aus. Damit sind die Arbeiten am Projekt auf .NET-Ebene erst einmal erledigt.

Entity Data Model-Code erstellen

In der Access-Datenbank, die Sie mit dem Code aus der Beispieldatenbank ausstatten, rufen wir nun den Befehl **EDMErstellen** aus dem Modul **mdlEDM** auf. Die mittlerweile optimierte Version dieser Prozedur speichert das Ergebnis direkt in der Zwischenablage. Sie brauchen nach dem Aufruf also nur noch zu Visual Studio zu wechseln, dort die Klasse **BestellverwaltungContext.vb** zu öffnen und dort ganz unten unter der Klasse **BestellverwaltungContext** den Inhalt der Zwischenablage einzufügen.

Die unteren paar Zeilen enthalten die **DbSet**-Deklarationen, die Sie ausschneiden und in die Klasse **BestellverwaltungContext** einfügen. Darunter finden Sie die benötigten Namespace-Verweise, die Sie ausschneiden und ganz oben über der Klassendefinition ergänzen können.

EDM: DataGrid als Datenblatt

Wer von Access kommt und Datenbankanwendungen mit Visual Studio programmieren möchte, vermisst vermutlich das einfache Datenblatt, das man unter Access mit wenigen Mausklicks zusammenstellen konnte. Dieses zeigt nicht nur die Datensätze der als Datenherkunft verwendeten Tabelle oder Abfrage an, sondern bietet auch die Möglichkeit, die enthaltenen Daten zu ändern, zu löschen oder durch neue Datensätze zu ergänzen. Dies wollen wir in diesem Artikel durch den Einsatz eines entsprechend programmierten DataGrid-Steuerelements nachbauen.

Beispieldaten

In den Artikeln **Von Access zu Entity Framework: Datenmodell** und **Von Access zu Entity Framework: Daten** haben wir gezeigt, wie Sie ein Entity Data Modell auf Basis eines Access-Datenmodells erstellen und die Daten für die daraus zu erzeugenden Tabellen in eine **Seed**-Methode schreiben. Im Beispielprojekt dieses Artikels haben wir das Entity Data Model und die Klasse mit der **Seed**-Methode bereits angelegt, sodass Sie nach dem Kopieren des Projektordners auf Ihren Rechner einfach nur den die Anweisung Update-Database in der Paket-Manager-Konsole ausführen müssen. Dadurch sollte automatisch eine Datenbank für den in der Verbindungszeichenfolge angegebenen SQL Server, hier **LocalDb**, angelegt und mit den angegebenen Tabellen und Daten gefüllt werden.

Fenster mit DataGrid anlegen

Wir wollen gleich ein eigenes Fenster für dieses Beispiel anlegen, das Sie dann über eine Schaltfläche vom Fenster **MainWindow** aus aufrufen können. Dazu legen wir die folgende Schaltfläche in diesem Fenster an:

```
<Grid>
    <Button x:Name="btnProdukteDataGrid" Click="btnProdukteDataGrid_Click">DataGrid mit Produkten</Button>
</Grid>
```

Die Schaltfläche soll die folgende Ereignismethode aufrufen:

```
Private Sub btnProdukteDataGrid_Click(sender As Object, e As RoutedEventArgs)
    Dim wnd As Window
    wnd = New ProdukteDataGrid
    wnd.ShowDialog()
End Sub
```

Das neue Fenster nennen wir **ProdukteDatagrid.xaml**. Zunächst legen wir ein paar Elemente in der Code behind-Klasse an, der wir zunächst die Konstruktor-Methode hinzufügen:

```
Public Class ProdukteDataGrid
    Public Sub New()
```

```
    End Sub
End Class
```

Wir benötigen einen Verweis auf den Namespace **System.Collections.ObjectModel**:

```
Imports System.Collections.ObjectModel
```

In der Klasse definieren wir eine lokale Variable für die Kontextklasse sowie die Liste der Produkte:

```
Private dbContext As BestellverwaltungContext
Private m_Produkte As ObservableCollection(Of Produkt)
```

Die Liste der Produkte wollen wir über eine öffentliche Eigenschaft namens **Produkte** verfügbar machen:

```
Public Property Produkte() As ObservableCollection(Of Produkt)
    Get
        Return m_Produkte
    End Get
    Set(ByVal value As ObservableCollection(Of Produkt))
        m_Produkte = value
    End Set
End Property
```

Damit das DataGrid beim Öffnen direkt mit den Daten gefüllt wird, erweitern wir die Konstruktormethode mit diesen Anweisungen:

```
Public Sub New()
    InitializeComponent()
    dbContext = New BestellverwaltungContext
    Produkte = New ObservableCollection(Of Produkt)(dbContext.Produkte)
    DataContext = Me
End Sub
```

Darin füllen wir die Kontext-Variable und die **List**-Variable **Produkte** mit der Liste der Produkte. Schließlich stellen wir den **DataContext** auf die Code behind-Klasse ein. Wenn wir nun das Projekt starten und im Fenster **MainWindow** auf die Schaltfläche klicken, erscheint das DataGrid wie in Bild 1.

Kombinationsfelder anzeigen

Die Daten der Felder **KategorieID** und **MehrwertsteuersatzID** möchten wir gern aus den jeweils verknüpften Auflistungen **Kategorien** und **Mehrwertsteuersaetze** füllen. Außerdem wollen wir noch die Spaltenüberschriften anpassen. Das erledigen wir mit den folgenden Erweiterungen, wobei wir erst die Daten für die Kombinationsfelder in der Code behind-Klasse verfügbar machen. Dazu fügen wir zwei private Variablen für die Listen der Kategorien und der Mehrwertsteuersätze hinzu:

```
Private m_Kategorien As List(Of Kategorie)
Private m_Mehrwertsteuersaetze As List(Of Mehrwertsteuersatz)
```

Für beide richten wir entsprechende öffentliche Eigenschaften ein:

```
Public Property Kategorien As List(Of Kategorie)
    Get
        Return m_Kategorien
    End Get
    Set(ByVal value As List(Of Kategorie))
        m_Kategorien = value
    End Set
End Property

Public Property Mehrwertsteuersaetze As List(Of Mehrwertsteuersatz)
    Get
        Return m_Mehrwertsteuersaetze
    End Get
    Set(ByVal value As List(Of Mehrwertsteuersatz))
        m_Mehrwertsteuersaetze = value
    End Set
End Property
```

Schließlich füllen wir diese in der Konstruktor-Methode aus den entsprechenden DbSets:

```
Public Sub New()
    ...
    Kategorien = New List(Of Kategorie)(dbContext.Kategorien)
    Mehrwertsteuersaetze = New List(Of Mehrwertsteuersatz)(dbContext.Mehrwertsteuersaetze)
    DataContext = Me
End Sub
```

Damit stehen die Daten der Kategorien und der Mehrwertsteuersätze über die öffentlichen Variablen **Kategorien** und **Mehrwertsteuersaetze** für den Zugriff aus der XAML-Seite bereit. Diese passen wir nun wie folgt an. Zunächst stellen wir **AutoGenerateColumns** auf **False** ein, da wir die Spalten nun von Hand definieren:

```
<DataGrid x:Name="dgProdukte" ItemsSource="{Binding
Produkte}" AutoGenerateColumns="False">
```

ProdukteDataGrid

ID	Name	KategorieID	Einzelpreis	MehrwertsteuersatzID	Bestand	Mindestbestand
1	Artikel 1	1	414	1	696	180
2	Artikel 2	2	544	1	815	541
3	Artikel 3	3	510	1	229	620
4	Artikel 4	4	681	2	887	371
5	Artikel 5	5	294	2	151	530
6	Artikel 6	6	585	1	364	876
7	Artikel 7	1	191	2	684	748
8	Artikel 8	2	782	2	168	808
9	Artikel 9	3	957	1	67	62
10	Artikel 10	4	380	2	464	120
11	Artikel 11	5	175	1	49	715
12	Artikel 12	6	561	2	218	469
13	Artikel 13	1	758	1	400	903

Bild 1: DataGrid mit den Daten der Tabelle **Produkte**

Dann folgen die einzelnen Spaltendefinitionen im **DataGrid.Columns**-Element:

```
<DataGrid.Columns>
    <DataGridTextColumn Binding="{Binding Path=Name}" Header="Produkt"></DataGridTextColumn>
```

Das Kombinationsfeld realisieren wir als **DataGridComboBocColumn**-Element, für das wir den Header festlegen sowie das gebundene Feld der Datenquelle des DataGrids beziehungsweise des Fensters (**SelectedValueBinding**), den Namen des anzuzeigenden Feldes (**DisplayMemberPath**) und den Namen des gebundenen Feldes (**SelectedValuePath**):

```
    <DataGridComboBoxColumn Header="Kategorie" SelectedValueBinding="{Binding KategorieID}"
        DisplayMemberPath="Name" SelectedValuePath="ID">
```

Die Datenquelle für das Kombinationsfeld müssen wir über einen Umweg festlegen, indem wir die Eigenschaft **ItemsSource** manuell definieren und füllen. Die Daten beziehen wir dabei aus der entsprechenden Eigenschaft des übergeordneten **Window**-Elements. Dies legen wir sowohl für die Eigenschaft **ElementStyle** als auch für die Eigenschaft **EditingElementStyle** fest. Dabei liefert **ElementStyle** die Formatierung für den angezeigten Text des Kombinationsfeldes und **EditingElementStyle** die Formatierung für die Elemente in der Liste nach dem Aufklappen:

```
        <DataGridComboBoxColumn.ElementStyle>
            <Style TargetType="{x:Type ComboBox}">
            <Setter Property="ItemsSource" Value="{Binding Path=DataContext.Kategorien,
                RelativeSource={RelativeSource AncestorType={x:Type Window}}}" />
        </Style>
        </DataGridComboBoxColumn.ElementStyle>
        <DataGridComboBoxColumn.EditingElementStyle>
            <Style TargetType="{x:Type ComboBox}">
                <Setter Property="ItemsSource" Value="{Binding Path=DataContext.Kategorien,
                    RelativeSource={RelativeSource AncestorType={x:Type Window}}}" />
            </Style>
        </DataGridComboBoxColumn.EditingElementStyle>
    </DataGridComboBoxColumn>
```

Beim Kombinationsfeld für die Mehrwertsteuersätze ist noch eine weitere Information in Form eines zusätzlichen Setters erforderlich, und zwar die Angabe der Formatierung. Während die Mehrwertsteuersätze in Form der Werte **0,07** und **0,19** gespeichert sind, sollen diese als **7,00%** und **19,00%** im Kombinationsfeld erscheinen. Daher definieren wir hier zusätzlich den Setter für die Eigenschaft **ItemStringFormat**:

```
    <DataGridComboBoxColumn Header="MwSt.-Satz" SelectedValueBinding="{Binding MehrwertsteuersatzID}"
        DisplayMemberPath="Name" SelectedValuePath="ID">
        <DataGridComboBoxColumn.ElementStyle>
            <Style TargetType="{x:Type ComboBox}">
```

```
                <Setter Property="ItemsSource" Value="{Binding Path=DataContext.Mehrwertsteuersaetze,
                    RelativeSource={RelativeSource AncestorType={x:Type Window}}}" />
                <Setter Property="ItemStringFormat" Value="P"></Setter>
            </Style>
        </DataGridComboBoxColumn.ElementStyle>
        <DataGridComboBoxColumn.EditingElementStyle>
            <Style TargetType="{x:Type ComboBox}">
                <Setter Property="ItemsSource" Value="{Binding Path=DataContext.Mehrwertsteuersaetze,
                    RelativeSource={RelativeSource AncestorType={x:Type Window}}}" />
                <Setter Property="ItemStringFormat" Value="P"></Setter>
            </Style>
        </DataGridComboBoxColumn.EditingElementStyle>
    </DataGridComboBoxColumn>
```

Schließlich folgen noch die Spaltendefinitionen für die übrigen Felder:

```
    <DataGridTextColumn Binding="{Binding Path=Einzelpreis}" Header="Einzelpreis"></DataGridTextColumn>
    <DataGridTextColumn Binding="{Binding Path=Bestand}" Header="Bestand"></DataGridTextColumn>
    <DataGridTextColumn Binding="{Binding Path=Mindestbestand}" Header="Mindestbestand"></DataGridTextColumn>
    <DataGridCheckBoxColumn Binding="{Binding Path=Auslaufprodukt}" Header="Auslaufprodukt"></DataGridCheckBoxColumn>
  </DataGrid.Columns>
</DataGrid>
```

Nach einem Neustart sieht das **DataGrid**-Steuerelement nun wie in Bild 2 aus.

Bearbeiten vorhandener Datensätze

Nun schauen wir uns an, wie der Benutzer die vorhandenen Datensätze bearbeiten und die Änderungen an den bearbeiteten Datensätzen speichern kann. Dazu bedarf es keiner großartigen Änderungen am Design des **DataGrid**-Steuerelements. Die Änderungen sind jetzt schon aktiviert – Sie können Änderungen an den Feldern durchführen, die auch nach dem Verlassen der Zeile beibehalten werden. Allerdings werden diese natürlich nicht in der zugrunde liegenden Tabelle gespeichert.

Geänderten Datensatz speichern

Dazu müssen wir noch einen entsprechenden Mechanismus einrichten. Als Erstes benötigen wir ein Ereignis, dass beim Verlassen der aktuell markierten Zeile ausgelöst wird. Dieses finden wir in der Ereignisprozedur **RowEditEnding**, welches wir dem **DataGrid**-Element wie folgt hinzufügen:

```
<DataGrid x:Name="dgProdukte" ItemsSource="{Binding Produkte}" AutoGenerateColumns="False" RowEditEnding="dgProdukte_Ro-
wEditEnding">
```

Dieses Ereignis hinterlegen wir in der Code behind-Datei von **ProdukteDataGrid.xaml** wie folgt und füllen es mit einer einzigen Anweisung, nämlich der folgenden:

```
Private Sub dgProdukte_RowEditEnding(sender As Object, e As DataGridRowEditEndingEventArgs)
    dbContext.SaveChanges()
End Sub
```

Diese speichert nach dem Auslösen die für den Datenbankkontext durchgeführten Änderungen in die Datenbank. Wenn Sie nun direkt die Anwendung starten und die Speichern-Funktion testen wollen, werden Sie keinen Erfolg haben: Noch wird dieses Ereignis nämlich nicht ausgelöst. Dazu müssen Sie erst alle Bindungen der zu ändernden Steuerelemente mit ein paar zusätzlichen Attributen versehen. Für die Textfelder sieht das etwa wie folgt aus:

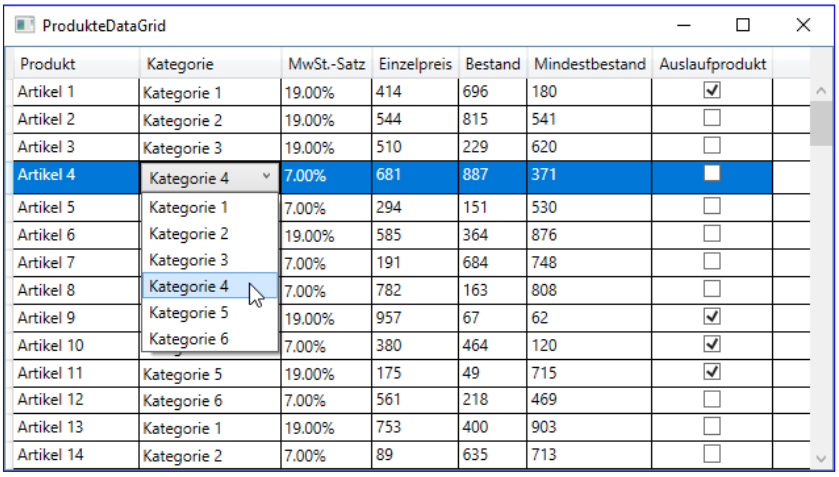


Bild 2: DataGrid mit Kombinationsfeldern

```
<DataGridTextColumn Binding="{Binding Path=Name, NotifyOnTargetUpdated=True, Mode=TwoWay, UpdateSourceTrigger=PropertyChanged}" Header="Produkt"></DataGridTextColumn>
```

Und dies ist die Änderung, die Sie für ein Kombinationsfeld durchführen müssen:

```
<DataGridComboBoxColumn Header="Kategorie" SelectedValueBinding="{Binding KategorieID, NotifyOnTargetUpdated=True, Mode=TwoWay, UpdateSourceTrigger=PropertyChanged}" DisplayMemberPath="Name" SelectedValuePath="ID">
```

Wir fügen also der **Binding**-Eigenschaft der Textspalten und der **SelectedValueBinding**-Eigenschaft der Kombinationsfeldspalte die drei Attribute **NotifyOnTargetUpdate**, **Mode** und **UpdateSourceTrigger** mit den angegebenen Werten hinzu. Anschließend können wir den Inhalt einer der Spalten für einen Datensatz ändern und die Änderungen werden auch in die Datenbank übertragen. Die **Binding**-Eigenschaften müssen Sie natürlich für alle Steuerelemente, die eine Spalte im **DataGrid**-Steuerele-

AUSGABE 06/2018

DATENBANK ENTWICKLER

MAGAZIN FÜR DIE DATENBANKENTWICKLUNG MIT VISUAL STUDIO FÜR DESKTOP, WEB UND CO.

TOP-THEMEN:

SQL SERVER	SQL Server-Datenbank ins Web mit SQL Azure	SEITE 3
EF	EDM: m:n-Beziehungen per Code First	SEITE 15
EF CORE	SQLite Code First mit EF Core	SEITE 19
C#-KLASSEN	C#-DLL in VB-Projekt nutzen	SEITE 28
POWERAPPS	Einstieg, Datenbanken, Artikel verwalten	SEITE 57

SQL Server-Datenbank ins Web mit SQL Azure

In einigen vorangehenden Artikeln haben wir gezeigt, wie Sie eine SQL Server-Datenbank auf Basis des Entity Data Models eines Projekts erstellen und unter einem Azure-Konto über eine Webseite veröffentlichen. In diesem Artikel wollen wir uns ansehen, wie Sie eine bestehende SQL Server-Datenbank ohne Entity Data Model als Azure-Datenbank anlegen. Auf diese Datenbank können Sie dann sowohl von Desktop-Anwendungen aus zugreifen als auch von PowerApps, die wir ebenfalls in dieser Ausgabe vorstellen.

Microsoft bietet die Möglichkeit, eigene SQL Server-Datenbanken über das Internet verfügbar zu machen. Dazu benötigen Sie ein Azure-Konto bei Microsoft, unter dem Sie dann einige teils kostenlose und teils kostenpflichtige Dienste in Anspruch nehmen können. Den Einstieg finden Sie unter dem folgenden Link:

```
https://azure.microsoft.
com/de-de/free/
```

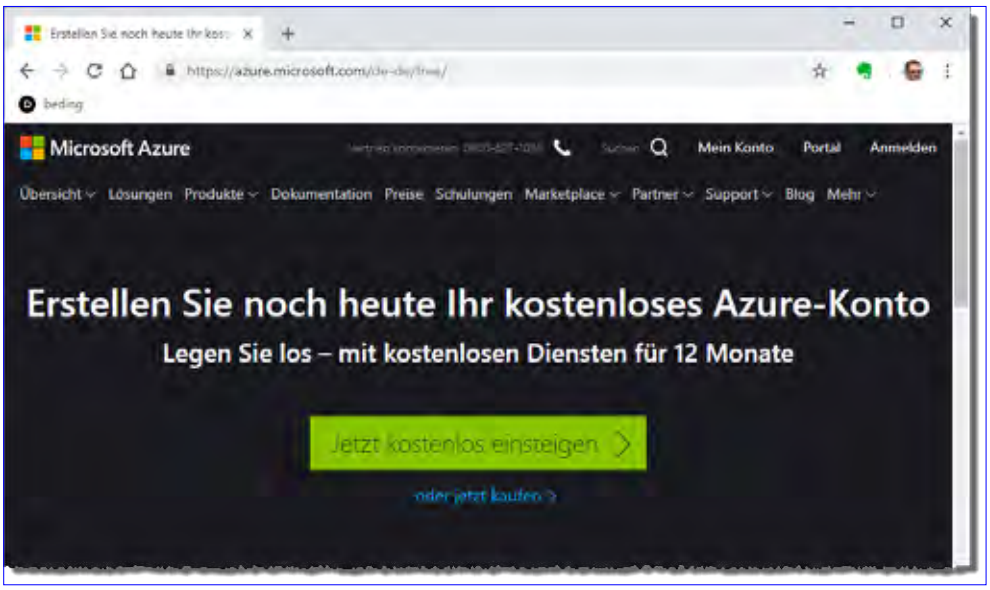


Bild 1: Einstieg mit dem kostenlosen Azure-Konto

Dieser führt Sie zu einer Seite, wo Sie ein zunächst kostenloses Konto anlegen können (siehe Bild 1). Eines vorneweg: Wenn Sie SQL Server-Datenbanken betreiben wollen, sind damit Kosten verbunden. Ich sehe keine Möglichkeit, einen solchen Dienst kostenlos in Anspruch zu nehmen – auch wenn Sie einen eigenen Server mieten und dort einen SQL Server betreiben, fallen dabei Kosten an.

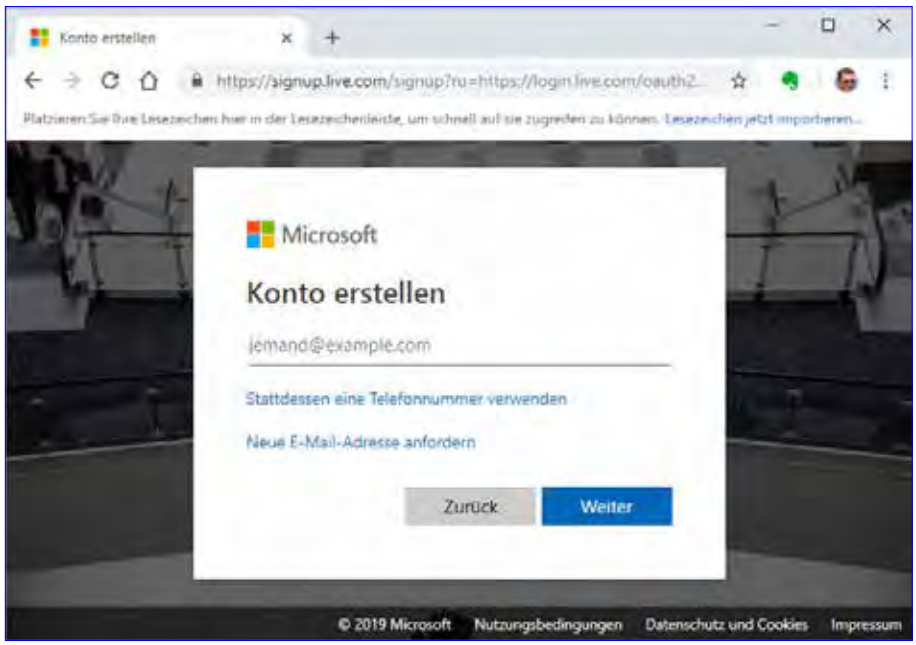


Bild 2: Anmelden bei live.com

Als Nächstes legen Sie nun ein Konto auf der Seite **live.com** an, wo Sie

ein paar Schritte später landen. Nach der Eingabe der E-Mail-Adresse auf der Seite aus Bild 2 geben Sie auch noch ein Kennwort an.

Danach fragt Microsoft noch ein paar weitere Daten ab wie das Land und das Geburtsdatum (siehe Bild 3).

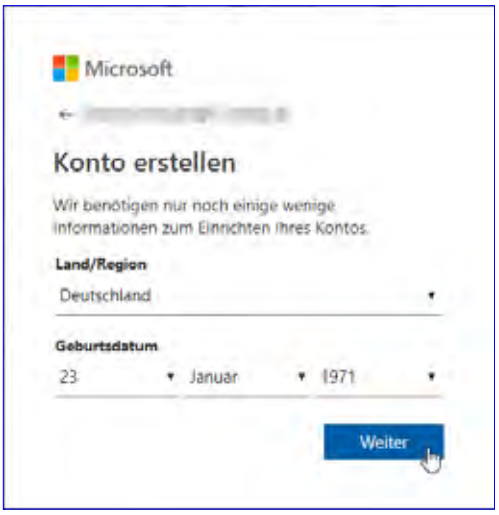


Bild 3: Weitere Schritte zum Erstellen des Microsoft-Kontos

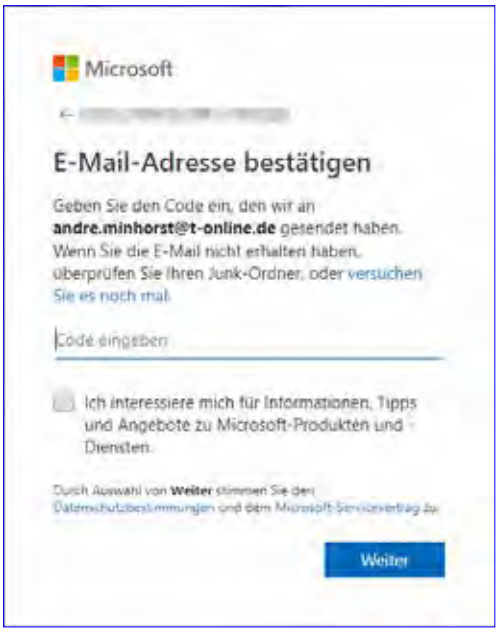


Bild 4: Abschluss der Anmeldung bei Microsoft

Anschließend müssen Sie die E-Mail-Adresse bestätigen, indem Sie einen an die angegebene Adresse gesendeten Code eingeben (siehe Bild 4). Schließlich folgt noch eine Prüfung, ob das Konto von einer realen Person erstellt wird und welche Telefonnummer zur Wiederherstellung des Kontos genutzt werden soll.

Azure-Konto registrieren

Danach folgt die Registrierung des Azure-Kontos. Dazu geben Sie im ersten Dialog die persönlichen Informationen an, also Land, Vorname, Nachname, E-Mail-Adresse und so weiter (siehe Bild 5). Hier geben Sie dann die E-Mail-Adresse ein, mit der Sie das Microsoft-Konto erstellt haben (wenn Sie nicht schon eines besessen haben).

Danach folgt eine Identitätsprüfung per Telefon, bei der eine Nachricht an die zuvor angegebene Telefonnummer geschickt wird. Nachdem Sie die Nachricht erhalten haben, geben Sie

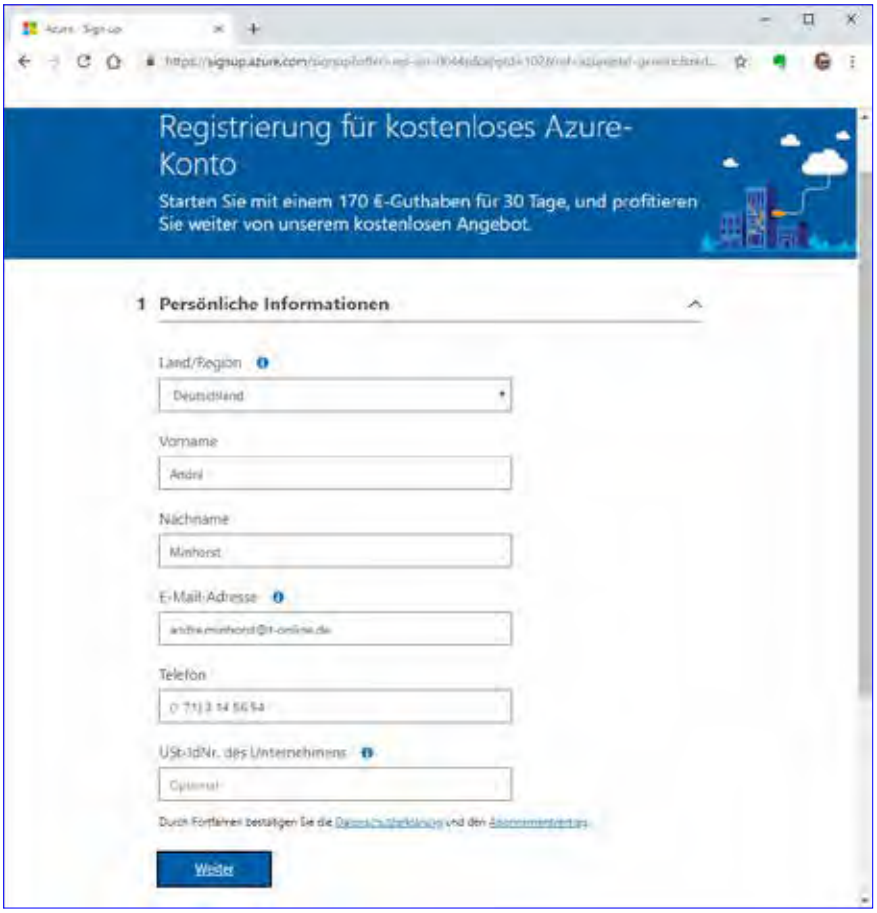


Bild 5: Registrierung für ein Azure-Konto

den enthaltenen Prüfcode in das entsprechende Textfeld ein. Im nächsten Schritt erfolgt eine weitere Identitätsüberprüfung über die Kreditkarte.

Diese wird nach der Angabe auf der Webseite erst nach einem Upgrade auf einen kostenpflichtigen Dienst belastet. Nachdem Sie die Kartendaten angegeben und einige Optionen wie die Datenschutzerklärung abgehakt haben, können Sie die Registrierung abschließen.

Das Azure-Portal

Danach gelangen Sie entweder direkt oder über den folgenden Link zum Azure-Portal, wo Sie sich gegebenenfalls noch einloggen müssen:

https://portal.azure.com/

Das Portal erwartet Sie mit der Übersicht aus Bild 6. Hier klicken Sie auf den Eintrag **SQL-Datenbanken**.

Damit gelangen Sie zum Bereich **SQL-Datenbanken**. Wenn Sie für Ihr Konto zuvor noch keine Datenbank erstellt haben, finden Sie hier die Meldung **Keine SQL-Datenbanken zum Anzeigen** vor.

Das ist kein Problem, denn direkt darunter erscheint der Link **SQL-Datenbank erstellen** (siehe Bild 7). Diesen klicken Sie an, um zum Dialog zum Erstellen einer neuen SQL-Datenbank zu gelangen.

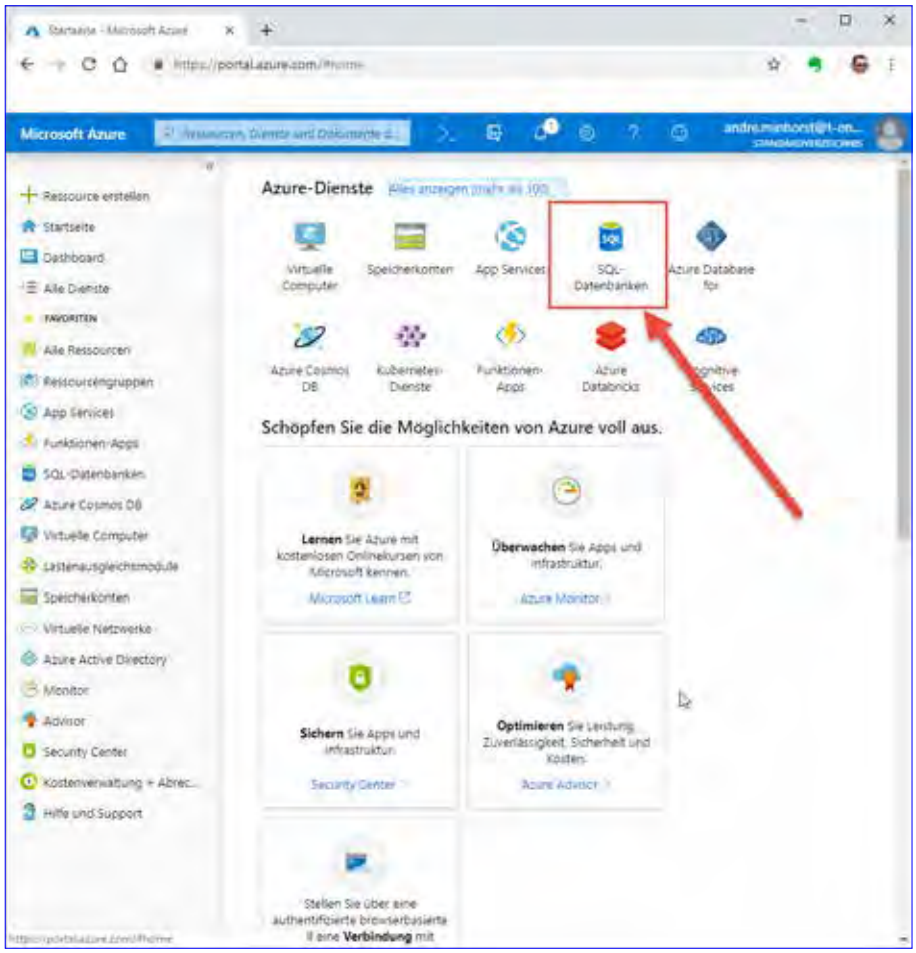


Bild 6: Das Azure-Portal

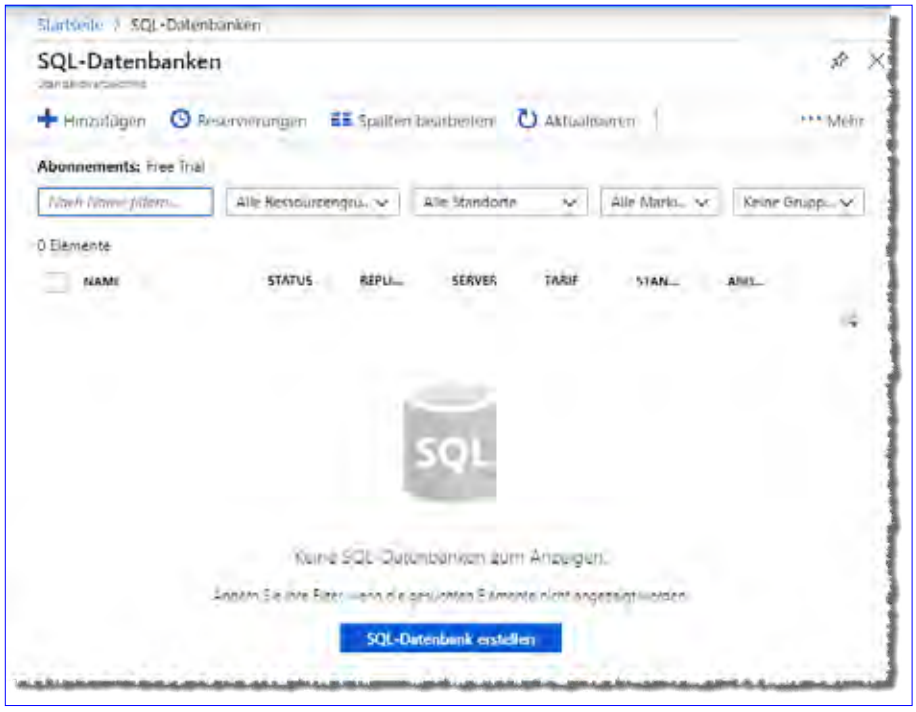


Bild 7: Dialog zum Verwalten der Datenbanken

Ressourcengruppe erstellen

Hier taucht gleich die erste Herausforderung auf: Es wird (bei einem neuen Account) zwar das Abonnement namens **Free Trial** voreingestellt, aber darunter findet sich die Option **Ressourcengruppe**, mit der wir zunächst einmal nichts anfangen können. Ein Klick auf das Auswahlfeld mit dem Text **Vorhandene auswählen...** führt nicht zum Erfolg, da hier bei einem neuen Konto noch keine Einträge vorhanden sind. Also verwenden wir den Link **Neues Element erstellen** direkt darunter (siehe Bild 8).

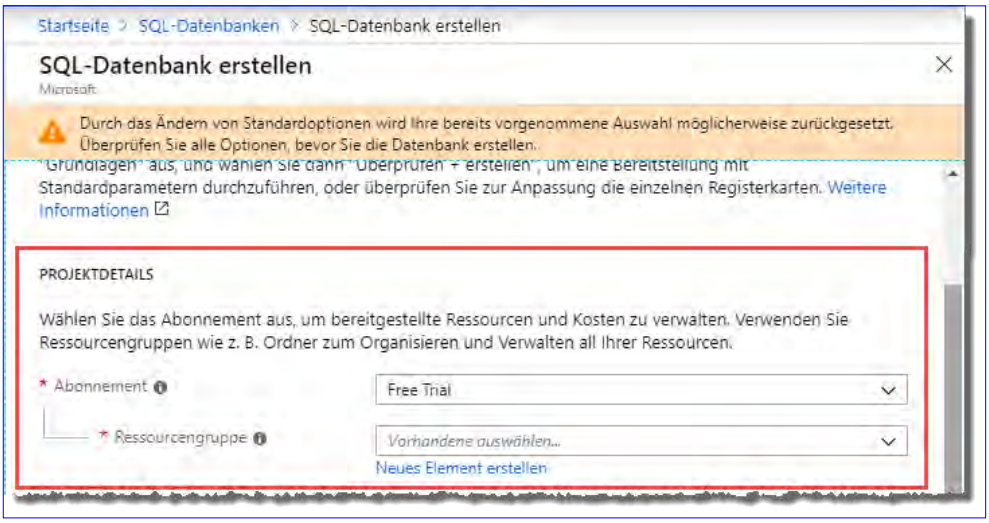


Bild 8: Anlegen einer neuen SQL-Datenbank

Die Ressourcengruppe ist eine Art Container, in der Sie verschiedene Ressourcen verwalten können, also etwa eine Webanwendung und eine SQL-Datenbank. Wir benötigen im Rahmen des aktuellen Artikels nur eine SQL-Datenbank für ein Beispiel, daher benennen wir die Ressourcengruppe einfach **Beispiel-SQL-Datenbank** (siehe Bild 9).

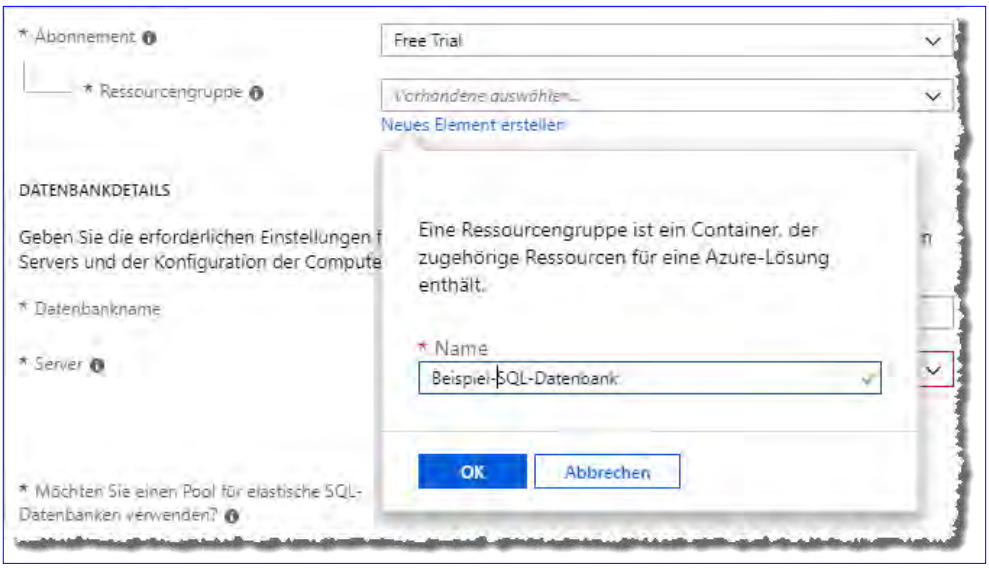


Bild 9: Anlegen einer neuen Ressource

Datenbankdetails festlegen

Nun können wir uns den weiter unten im Dialog einzugebenden Informationen

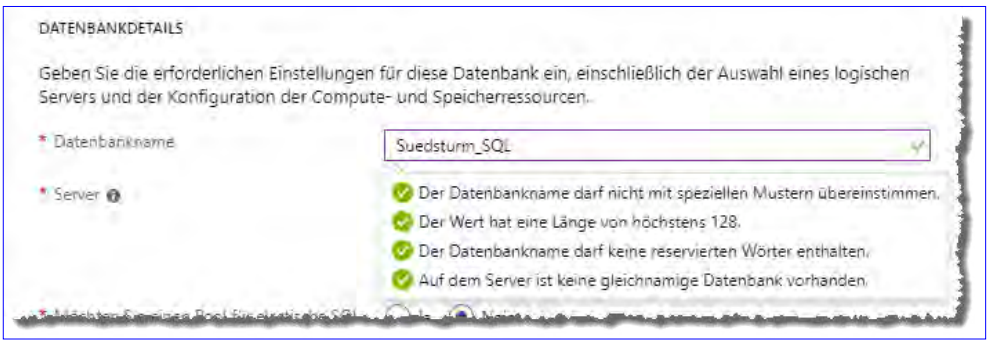


Bild 10: Eingabe des Datenbanknamens

zuwenden. Hier benötigen wir einen Datenbanknamen und einen Server. Der Datenbankname ist schnell gefunden – wir wollen die Südsturm-Datenbank, die wir in weiteren Artikeln verwenden, als Azure-Datenbank erstellen. Als Abgrenzung zur ursprünglich unter Access verwendeten Südsturm-Datenbank fügen wir hier noch den Zusatz **SQL** hinzu, also **Suedsturm_SQL**. Bezüglich des Datenbanknamens gibt es einige Einschränkungen, die aber bei der Eingabe eingeblendet werden (siehe Bild 10).

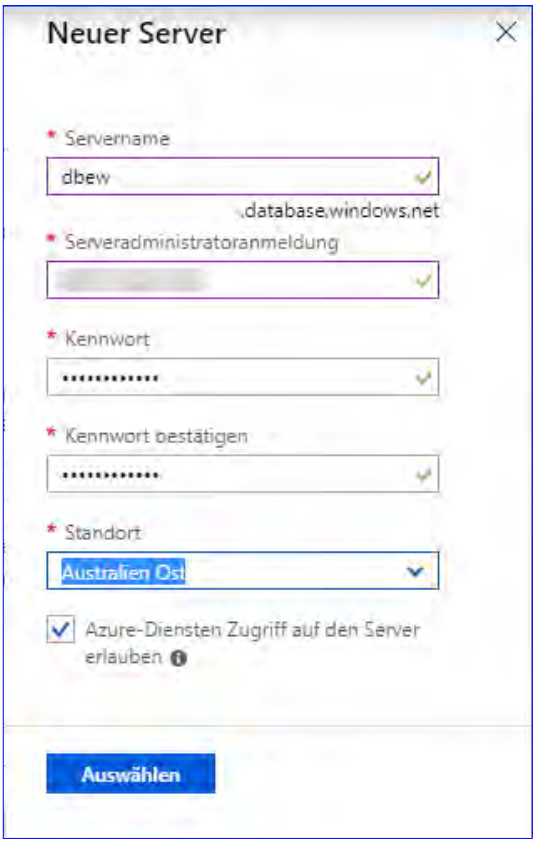


Bild 11: Anlegen eines neuen Servers

Server erstellen

Auch die Option **Server** bietet den Eintrag **Wählen Sie einen Server aus**, allerdings ist auch dieses Auswahlfeld leer. Also betätigen wir auch hier den Link **Neu erstellen**. Es erscheint ein Popup, mit dem wir den Servernamen, den Namen für die Administratoranmeldung und das Kennwort eingeben können. Als Standort wurde **Australien Ost** angegeben, was sich in unserem Fall auch nicht ändern ließ. Also haben wir den Wert so beibehalten. Nach dem Eintragen der Informationen legen Sie den neuen Server mit einem Klick auf **Auswählen** an (siehe Bild 11). Danach dauert es einige Sekunden, bis die Datenbank erstellt wurde.

Kosten im Griff

Wichtig ist noch, die richtige Datenbankversion zu wählen. Klicken Sie auf **Datenbank konfigurieren**, erscheint rechts ein neuer Bereich, der die verschiedenen Versionen (Basic, Standard und Premium) anbietet (siehe Bild 12). Üblicherweise ist **Standard** aktiviert. Für ein Beispiel benötigen Sie aber vermutlich kaum mehr als zwei Gigabyte Speicher, also wechseln wir zu **Basic**. Damit entstehen Kosten von mindestens 4,63 EUR pro Monat, wobei wir beim Erstellen unseres Azure-Accounts eine Gutschrift von 170 EUR erhalten haben – Sie können damit also theoretisch eine Weile experimentieren. Allerdings stehen Ihnen die 170

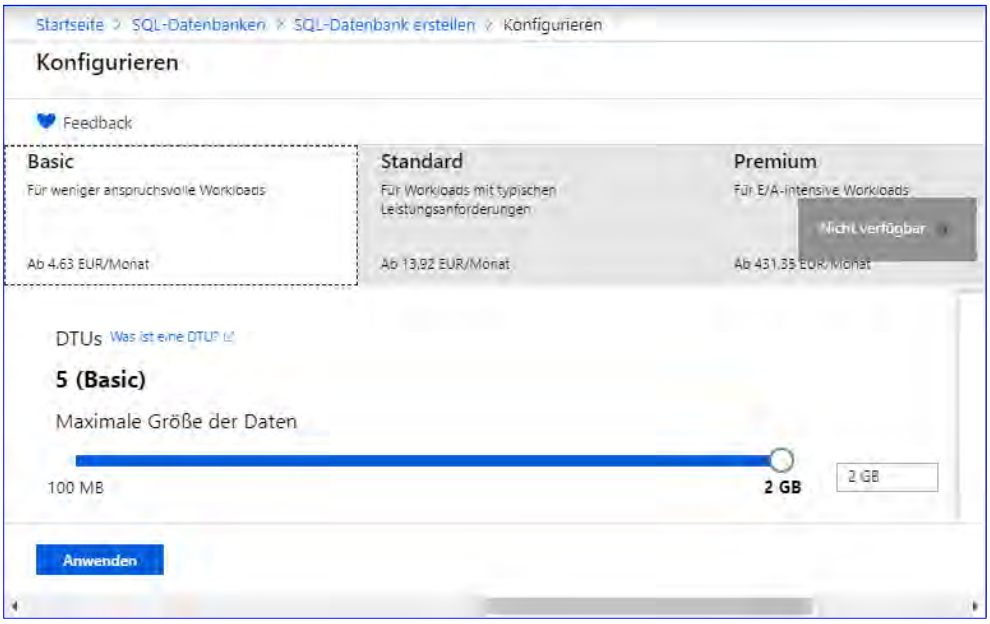


Bild 12: Auswahl des Preismodells

EDM: m:n-Beziehungen per Code First

Genau wie 1:n-Beziehungen können Sie mit Code First auch m:n-Beziehungen abbilden. Wer denkt, dass dazu wie in Datenbanksystem eine Art Verknüpfungsklasse notwendig ist, hat Recht – diese ist aber nur in bestimmten Fällen notwendig. Welche das sind und wie Sie die verschiedenen Arten einer m:n-Beziehung unter Code First abbilden, erfahren Sie in diesem Artikel. Außerdem zeigen wir, welche Datenmodelle beim Migrieren der Klassen in ein Datenbanksystem entstehen.

Vorbereitungen

Die Vorbereitungen laufen wie im Artikel **EDM: 1:n-Beziehungen per Code First** beschrieben.

Verschiedene Arten von m:n-Beziehungen

Wir kümmern uns in diesem Artikel um zwei verschiedene Arten von m:n-Beziehungen. Die erste Art geht davon aus, dass nur die Beziehung zwischen den Elementen zweier Klassen abgebildet werden soll. Das kann zum Beispiel die Beziehung zwischen Artikeln und Kategorien sein. Sie können jedem Artikel eine oder mehrere Kategorien zuweisen und jeder Kategorie ein oder mehrere Artikel. Dabei sind für eine Beziehung keine weiteren Informationen notwendig. Die zweite Art der Beziehung enthält zusätzliche Informationen zu jeder Kombination der beteiligten Klassen. Dabei kann es sich etwa die Beziehung zwischen einer Bestellung und den darin enthaltenen Artikeln handeln. Jede Bestellung kann mehrere Artikel enthalten und umgekehrt – insofern unterscheidet sich dies nicht von der einfachen m:n-Beziehung. Allerdings wollen wir in diesem Fall zu jeder Kombination noch weitere Informationen speichern, im einfachsten Fall die Menge der bestellten Artikel in einer Bestellung. Hier würden wir im Datenbanksystem einfach ein weiteres Feld zu der Verknüpfungstabelle hinzufügen. In der Modellierung der Klassen ergeben sich weitreichendere Unterschiede, wie die folgenden Beispiele zeigen werden.

Klassen für eine einfache m:n-Beziehung

Wir wollen zunächst eine einfache m:n-Beziehung abbilden, wie sie zwischen Artikeln und Kategorien vorgenommen wird. Dazu legen wir als Erstes die einfachen Klassen für diese beiden Entitäten an – in unserem Beispiel im Verzeichnis **Data**. Die Klasse enthält die Felder **ID**, **Name** und **Price**:

```
namespace EDMFluentAPI.Data {
    public class Article {
        public int ID { get; set; }
        public string Name { get; set; }
        public decimal Price { get; set; }
    }
}
```

Die Klasse für die Kategorien halten wir noch einfacher – sie soll nur das Feld **ID** und das Feld **Name** enthalten:

```
public class Category {
    public int ID { get; set; }
```

```
    public string Name { get; set; }
}
```

Wie stellen wir nun die Beziehung her? Das gelingt ganz einfach, indem wir beiden Klassen jeweils eine **ICollection**-Auflistung mit dem Typ des jeweils zu verknüpfenden Elements hinzufügen. Zunächst für die Klasse **Article**:

```
public class Article {
    ...
    public ICollection<Category> Categories { get; set; }
}
```

Für die Klasse **Category** sieht das andersherum aus:

```
public class Category {
    ...
    public ICollection<Article> Articles { get; set; }
}
```

Wir dürfen natürlich nicht vergessen, beide Klassen über entsprechende **DbSet**-Auflistungen zum Datenbankkontext hinzuzufügen:

```
public class CustomerManagementContext : DbContext {
    ...
    public virtual DbSet<Article> Articles { get; set; }
    public virtual DbSet<Category> Categories { get; set; }
    ...
}
```

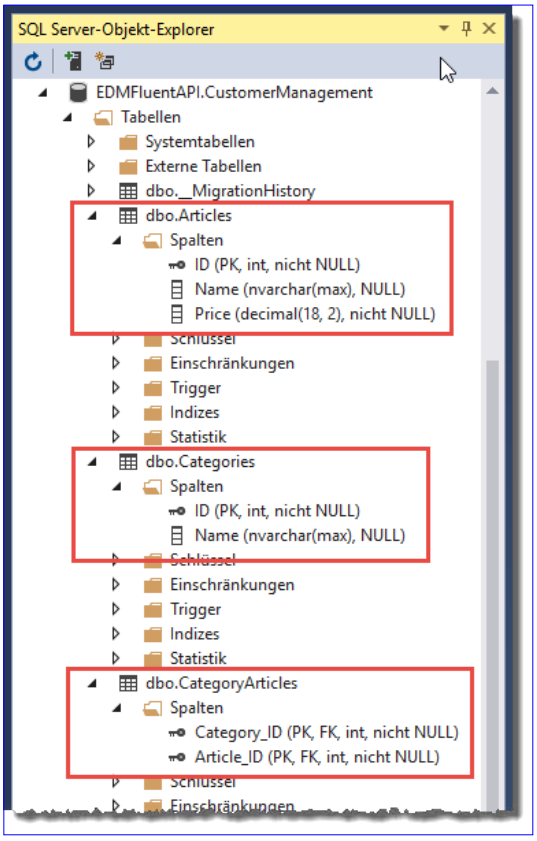


Bild 1: Anlegen einer einfachen m:n-Beziehung

Datenmodell erstellen

Wie überführt Entity Framework diese Angaben in das Datenmodell? Mit der Anweisung **add-migration AddArticlesCategories** in der Paket-Manager-Konsole erstellen wir das Skript für die Erstellung von Tabellen, Feldern und Beziehungen. Dieses wird direkt angezeigt, damit Sie es kontrollieren können. Dann führen Sie es mit der Anweisung **update-database** aus und schauen uns uns das Ergebnis im SQL Server-Objekt-Explorer an. Hier finden wir nun gleich drei neue Tabellen vor. Die Tabellen **Articles** und **Categories** kennen wir, die Tabelle **CategoryArticles** ist verknüpft die Tabellen **Articles** und **Categories** über die beiden Fremdschlüsselfelder **Category_ID** und **Article_ID**, welche jeweils mit den Tabellen **Categories** und **Articles** verknüpft sind (siehe Bild 1). Die Beziehung wurde sogar direkt mit einem zusammengesetzten Primärschlüssel erzeugt, sodass die zumeist vorliegende Anforderung, dass jede Kombination zweier Datensätze nur einmal in der Verknüpfungstabelle vorkommen darf, erfüllt ist.

Beziehung mit Verknüpfungsinformationen

Nun schauen wir uns den anderen Fall an, bei dem wir der Verknüpfung weitere Informationen hinzufügen wollen – in diesem Fall die Menge eines Artikels in einer Bestellung. Dazu legen wir zunächst wieder die beiden Klassen an, die durch die Bezie-

SQLite Code First mit EF Core

Wir haben bereits einige Male SQLite als Datenbanksystem verwendet. Allerdings hatten wir damals jeweils eine Datenbank mit SQLite erstellt und dann ein Entity Data Model dafür erstellt. In den letzten Ausgaben haben wir uns jedoch in Zusammenhang mit SQL Server-Datenbanken mit Code First-Migrationen befasst, also den Aufbau der Datenbank durch Klassen beschrieben und dann erst die Datenbank erstellen lassen. Das wollen wir nun auch mit SQLite machen, was erstmals durch Entity Framework Core möglich ist. Allerdings sind dazu ein paar Schritte mehr nötig, als es mit dem SQL Server der Fall ist.

Die .NET Core-Vorlagen sind noch überschaubar. Derzeit gibt es für den Desktop noch gar keine Vorlage, sondern nur für Konsolen, Klassenbibliotheken und etwa für ASP.NET-Anwendungen (aktueller Stand: **.NET Core 2.1**).

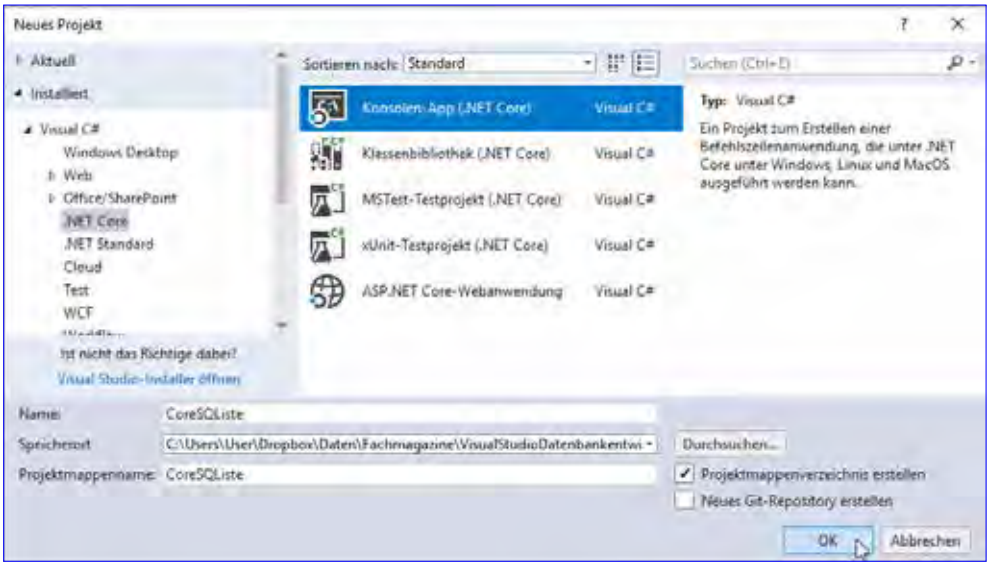


Bild 1: Anlegen eines Projekts auf Basis von .NET Core

Letztere haben wir ja schon programmiert, nun machen wir uns mit den anderen beiden Typen vertraut. Zum Testen verwenden wir zunächst die Vorlage **Konsolen-App (.NET Core)** – siehe Bild 1. Nachdem wir die Konsolen-App erstellt haben, finden wir die dort übliche Klassenbibliothek **Program.cs** vor. Und Sie sehen es richtig: Für diese Funktion, also den Zugriff per Entity Framework auf eine SQLite-Datenbank, können wir aktuell nicht nur nicht mit WPF arbeiten, sondern die Konsolen-App bietet auch noch keine Unterstützung der Programmiersprache Visual Basic.

Zum Glück haben wir uns in den ersten Ausgaben von **DATENBANKENTWICKLER** auch mit C# beschäftigt, sodass wir einige wenige Codezeilen leicht damit programmieren können. Um die Unterstützung für das Entity Framework hinzuzufügen, klicken Sie mit der rechten Maustaste auf das Projekt-Element im Projektmappen-Explorer und wählen aus dem dann erscheinenden Kontextmenü den Eintrag **NuGet-Pakete verwalten** aus. Hier wechseln Sie dann zum Bereich **Durchsuchen** und geben als Suchbegriff **EntityFrameworkCore.Sqlite** ein.

Wählen Sie hier die stabilste aktuelle Version aus und klicken Sie auf **Installieren** (siehe Bild 2). Danach werden Sie noch aufgefordert, der Lizenz zuzustimmen.

Den gleichen Vorgang wiederholen Sie anschließend für das Paket **Microsoft.EntityFrameworkCore.Design**. Schließlich fügen wir noch das Package **Microsoft.EntityFrameworkCore.Tools** hinzu.

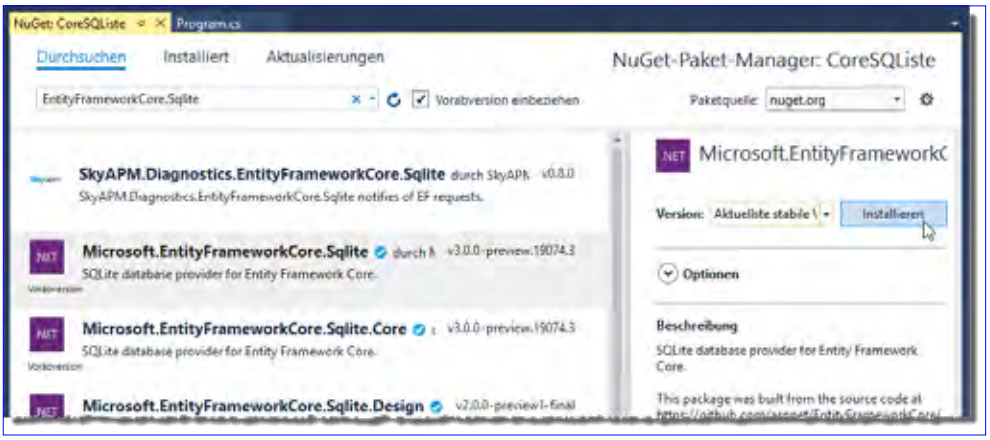


Bild 2: Hinzufügen der NuGet-Pakete

Modell erstellen

Nun benötigen wir das Entity Data Model. Dazu legen Sie eine neue Klasse namens **BestellverwaltungContext.cs** an. Dieser fügen wir zunächst einige Verweise auf Namespaces hinzu:

```
using System;
using System.ComponentModel.DataAnnotations.Schema;
using System.ComponentModel.DataAnnotations;
using Microsoft.EntityFrameworkCore;
```

Der Code der Klasse selbst sieht wie folgt aus. Die Klasse implementiert die Schnittstelle **DbContext**.

Als Erstes fügen wir eine Methode namens **OnModelCreating** hinzu, welche die eindeutigen Indizes für die betroffenen Felder hinzufügt:

```
namespace CoreSQListe {
    class BestellverwaltungContext : DbContext {
        //eindeutigen Index für EntityFrameworkCore:
        protected override void OnModelCreating(ModelBuilder builder) {
            builder.Entity<Anrede>()
                .HasIndex(a => a.Name)
                .IsUnique(true);
            builder.Entity<Bestellposition>()
                .HasIndex(b => new { b.BestellungID, b.ProduktID })
                .IsUnique(true);
        }
```

Die nächste Methode heißt **OnConfiguring** und legt den Namen der zu erstellenden beziehungsweise zu verwendenden SQLite-Datenbank fest. Das erledigen wir mit der Methode **UseSqlite** des mit dem Parameter **optionsBuilder** übergebenen Elements.

Der Parameter dieser Methode heißt **Data Source** und soll in diesem Fall den Wert **Bestellverwaltung.db** annehmen:

```
        protected override void OnConfiguring(DbContextOptionsBuilder optionsBuilder) {
            optionsBuilder.UseSqlite("Data Source=Bestellverwaltung.db");
        }
```

Nun folgenden die Deklarationen der **DbSet**-Elemente für die einzelnen Auflistungen beziehungsweise Tabellen:

```
        public DbSet<Anrede> Anreden { get; set; }
        public DbSet<Bestellposition> Bestellpositionen { get; set; }
        public DbSet<Bestellung> Bestellungen { get; set; }
        public DbSet<Kategorie> Kategorien { get; set; }
        public DbSet<Kunde> Kunden { get; set; }
        public DbSet<Mehrwertsteuersatz> Mehrwertsteuersaetze { get; set; }
        public DbSet<Produkt> Produkte { get; set; }
    }
```

Im Anschluss an die Definition der **DbContext**-Klasse legen wir die Klassen für die einzelnen Entitäten an. Diese erhalten jeweils mit der DataAnnotation den Namen der zu erstellenden Tabelle. Außerdem werden mit weiteren DataAnnotations wie **StringLength**, **Required** et cetera weitere Eigenschaften für die zu erstellenden Felder angegeben. Wir schauen uns an dieser Stelle nur die Definition der beiden Entitäten **Anrede** und **Bestellposition** an:

```
    [Table("Anreden")]
    public class Anrede {
        public int Id { get; set; }
        [StringLength(255)]
        [Required]
        public string Name { get; set; }
    }
    [Table("Bestellpositionen")]
    public class Bestellposition {
        public int ID { get; set; }
        [Required]
        public int BestellungID { get; set; }
        [Required]
        public int ProduktID { get; set; }
        [Required]
        public decimal Einzelpreis { get; set; }
        [Required]
        public decimal Mehrwertsteuersatz { get; set; }
        [Required]
        public decimal Rabatt { get; set; }
        [Required]
```

C#-DLL in VB-Projekt nutzen

Einige Techniken werden von Microsoft erst für die Benutzung unter C# bereitgestellt. Das ist aktuell etwa der Fall für einige der Core-Produkte wie Entity Framework Core. Wenn Sie diese dennoch frühzeitig nutzen wollen, aber normalerweise mit Visual Basic arbeiten, können Sie das auf folgende Art erledigen: Sie bauen eine DLL auf Basis der Vorlage für C# und binden diese dann in das Visual Basic-Projekt, in dem Sie die Hauptarbeit verrichten. Auf die Objekte, Eigenschaften und Methoden der C#-Bibliothek können Sie dann einfach zugreifen.

Um zu zeigen, wie der Einsatz etwa einer C#-DLL innerhalb einer Visual Basic-Konsolenanwendung funktioniert, erstellen wir zunächst ein Projekt des Typs **Visual BasicIKonsolen-App** (siehe Bild 1). Diese App nennen wir **VisualBasicKonsole**.

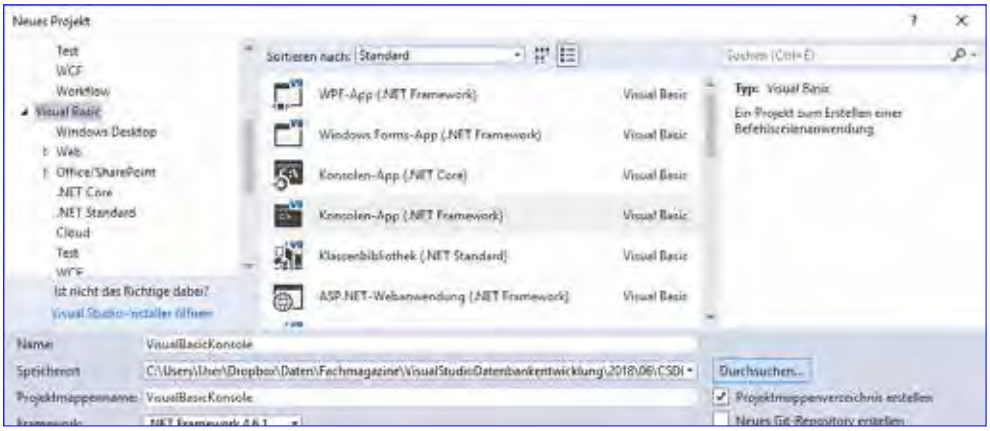


Bild 1: Visual Basic-Konsolenanwendung erstellen

Danach markieren wir im Projektmappen-Explorer den Eintrag mit der Projektmappe und wählen aus dem Kontextmenü den Befehl **HinzufügenINeues Projekt...** aus (siehe Bild 2).

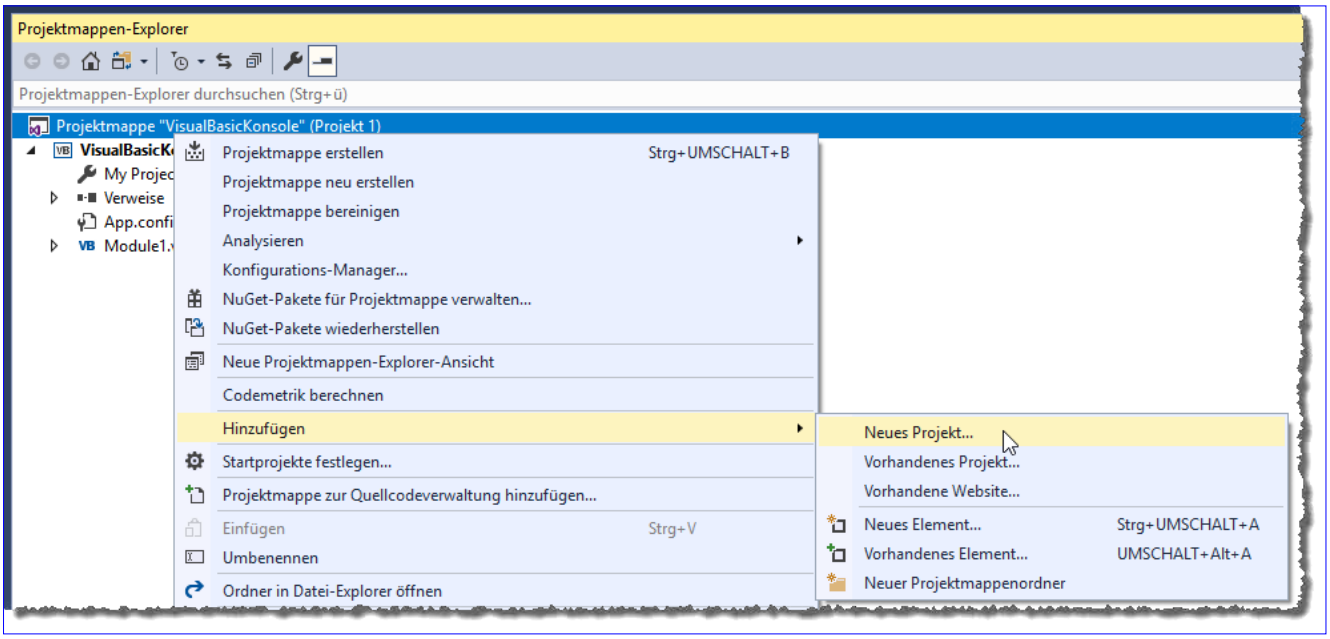


Bild 2: Neues Projekt zu Projektmappe hinzufügen

Im nun erscheinenden Dialog **Neues Projekt hinzufügen** wählen wir die Vorlage **Visual C#lKlassenbibliothek** aus (siehe Bild 3) und nennen diese **CSKlassenbibliothek**.

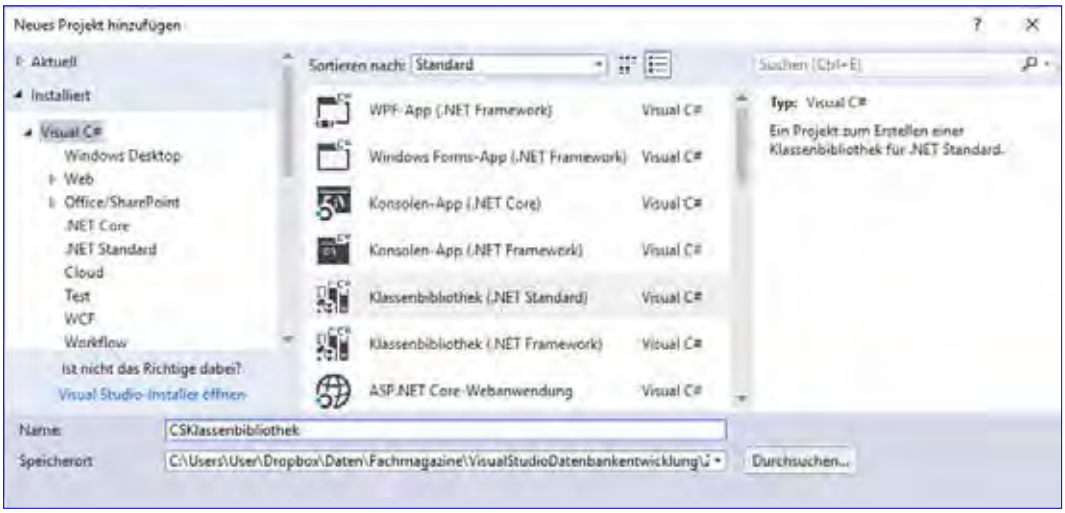


Bild 3: Projekttyp für zusätzliches Projekt festlegen

Im Projektmappen-Explorer sehen wir nun zwei Projekte, von denen eines in fetter Schrift dargestellt ist (das zuerst angelegte) und eines in normaler Schrift (siehe Bild 4).

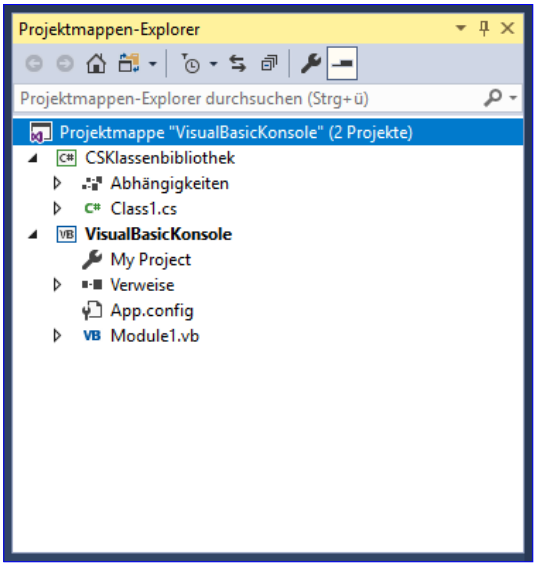


Bild 4: Projektmappe mit zwei Projekten

Das Projekt in fetter Schrift ist allerdings nur zufällig das zuerst angelegte: Eigentlich markiert die fette Schrift nämlich das Startprojekt, also das Projekt, das beim Start ausgeführt wird. Sie können ein anderes Projekt als Startprojekt auswählen, indem Sie den Kontextmenü-Eintrag **Als Startprojekt festlegen** eines aktuell nicht als Startprojekt festgelegten Projekts auswählen.

Verweis auf DLL hinzufügen

Damit Sie nun vom Startprojekt, also **VisualBasicKonsole**, auf die Elemente der DLL **CSKlassenbibliothek** zugreifen können, fügen Sie einen Verweis auf die DLL zur Konsolenanwendung hinzu. Dazu wählen Sie den Kontextmenü-Eintrag **Verweis hinzufügen...** des Eintrags **Verweise** des Projekts **VisualBasicKonsole** aus (siehe Bild 5).

Es erscheint der Dialog **Verweis-Manager**. Dieser zeigt praktischerweise direkt das andere Projekt der aktuellen Projektmappe im Bereich **Projekte\Projektmappe** an.

Also markieren wir den Eintrag **CSKlassenbibliothek** und schließen den Dialog mit der **OK**-Schaltfläche (siehe Bild 6).

Nach dem Schließen des Dialogs **Verweis-Manager** erscheint der neu hinzugefügte Verweis auch direkt unter dem Ordner **Verweise** der Konsolenanwendung (siehe Bild 7).

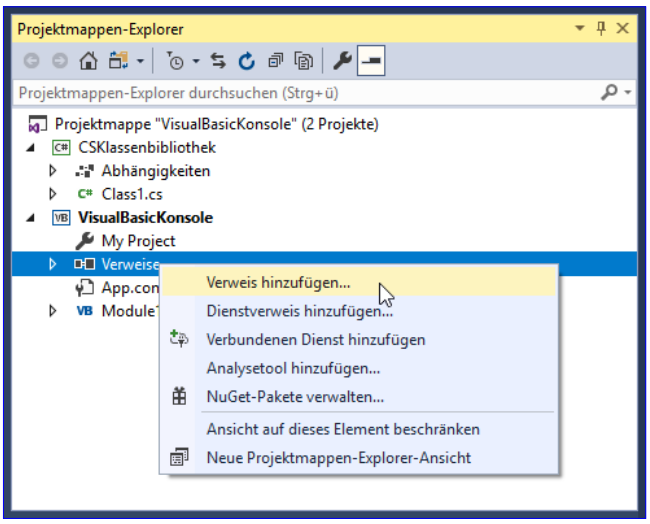


Bild 5: Verweis zu Projekt hinzufügen

Einstieg in PowerApps

PowerApps sind eine neue Möglichkeit, Anwendungen für Web, Desktop und mobile Geräte zu programmieren. Dabei geht es weniger darum, native Apps etwa für ein iPhone oder iPad zu programmieren. Stattdessen wählt Microsoft den Ansatz, eine Art Container-App zur Verfügung zu stellen, mit der man dann speziell für diese programmierte Anwendungen öffnen kann. Das gelingt allein über eine recht einfache Online-Entwicklungsumgebung. Aber es gibt auch Einschränkungen: Zum Beispiel kann nicht jeder eine solche App im App-Store erwerben. Wer die App nutzen will, muss die PowerApps-App auf seinem Gerät haben und muss für die Nutzung der App freigeschaltet sein. Dieser Artikel zeigt die Grundlagen rund um die Erstellung von PowerApps.

Voraussetzungen

Die erste Voraussetzung für die kommerzielle Nutzung der PowerApps oder die Nutzung in Teams ist das Vorhandensein einer Lizenz für Office 365, Dynamics 365 oder Power Apps Plan 1 oder 2.

Für die beiden letzteren gibt es kostenlose Testversionen, von denen wir in diesem Artikel die Variante **PowerApps Plan 1** nutzen wollen (zu finden unter **https://powerapps.microsoft.com/de-de/pricing/**).

Zum Glück gibt es wie von Visual Studio und SQL Server auch hier wieder eine Community-Version, die Sie hier aufrufen können: **https://powerapps.microsoft.com/de-de/communityplan/**

Bild 1: Einstieg in die Community-Version der PowerApps

Wir schauen uns diese Version der PowerApps an und klicken gleich auf **Steigen Sie kostenlos ein** (siehe Bild 1). Danach melden Sie sich mit einem Microsoft-Konto an oder erstellen ein neues. Nach ein paar weiteren organisatorischen Schritten stehen Sie vor der ersten Entscheidung: Sie können dann verschiedene Vorlagen für das Erstellen einer eigenen App nutzen. Es gibt die folgenden Möglichkeiten:

- **Canvas-App ohne Vorlage**: Hier entwerfen Sie die komplette App und verknüpfen diese mit den gewünschten Datenquellen.
- **Modellgesteuerte Apps ohne Vorlage**: Sie erstellen ein Modell Ihrer Daten und lassen dann auf Basis dieses Modells eine PowerApp erstellen.

- **Mit Daten beginnen**: Dies erstellt eine App auf Basis von Daten, die aus drei Bildschirmen besteht, die sie anschließend anpassen können.

- **PowerApps Training for Office**

Eine Canvas-App ohne Vorlage erstellen

Wir wollen zuerst die erste Vorlage ausprobieren. Dazu fahren Sie mit der Maus über das Element und wählen zunächst aus, ob die App für Smartphones oder für Tablets optimiert werden soll. Wir wählen Smartphone, da dieses im Gegensatz zu einem Tablet bei jedem Leser vorhanden sein sollte. Danach klicken Sie auf den Link **Diese App erstellen** (siehe Bild 2).

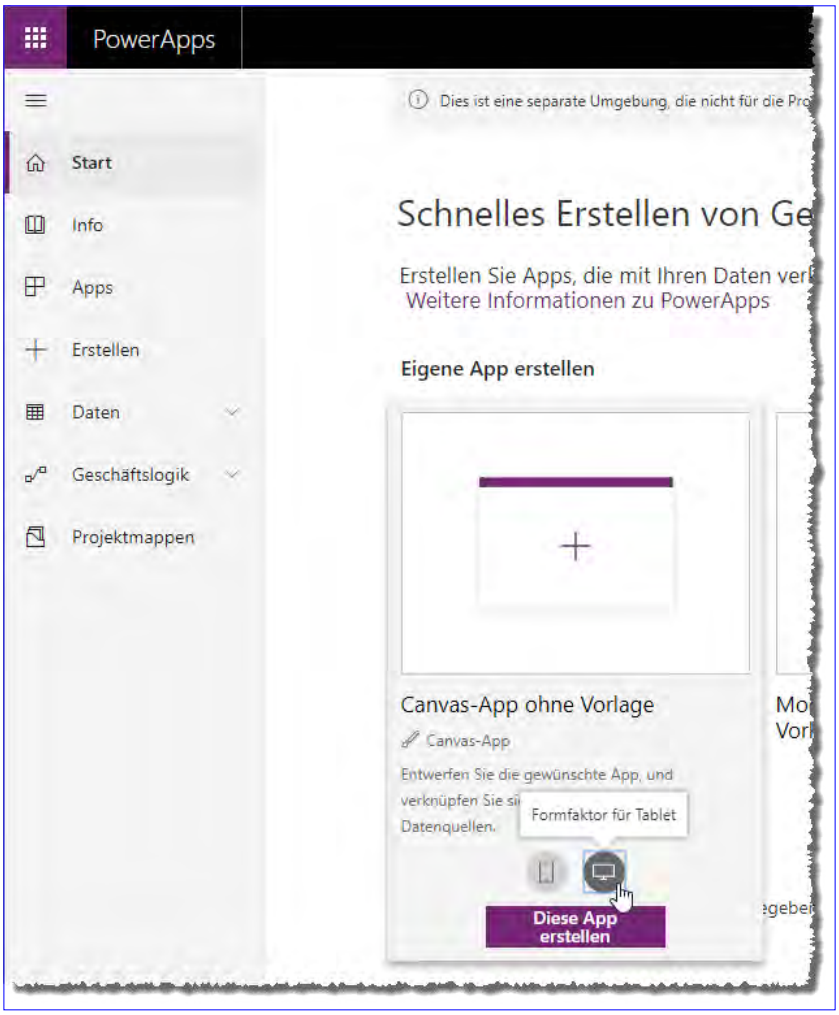


Bild 2: Erstellen einer Canvas-App ohne Vorlage

Danach öffnet sich die PowerApps-Entwicklungsumgebung im Browser. Hier können Sie noch auswählen, ob Sie ein Formular erstellen wollen oder einen Katalog. Außerdem finden Sie hier eine interaktive Tour (siehe Bild 3).

Formular erstellen bedeutet, eine Detailansicht von einem Datensatz anzuzeigen. **Katalog erstellen** heißt, eine Übersicht mit mehreren Datensätzen zu bauen.

Bild 3: Möglichkeiten beim Anlegen der App

Wir wollen erst einmal eine Anwendung nach einer Vorlage erstellen, um uns grundlegende Schritte anzusehen. Dazu nutzen wir die Option **Überspringen**, um die App direkt ohne Unterstützung zu erstellen und zu weiteren Vorlagen zu gelangen. Nun erscheint die Entwicklungsoberfläche für PowerApps (siehe Bild 4). Hier klicken wir links oben auf die Schaltfläche **Datei**.

Klicken Sie diesen Befehl an, erscheint ein neuer Bereich, der verschiedene Möglichkeiten zum Erstellen einer PowerApp anbietet – und diesmal etwas konkreter mit der Angabe

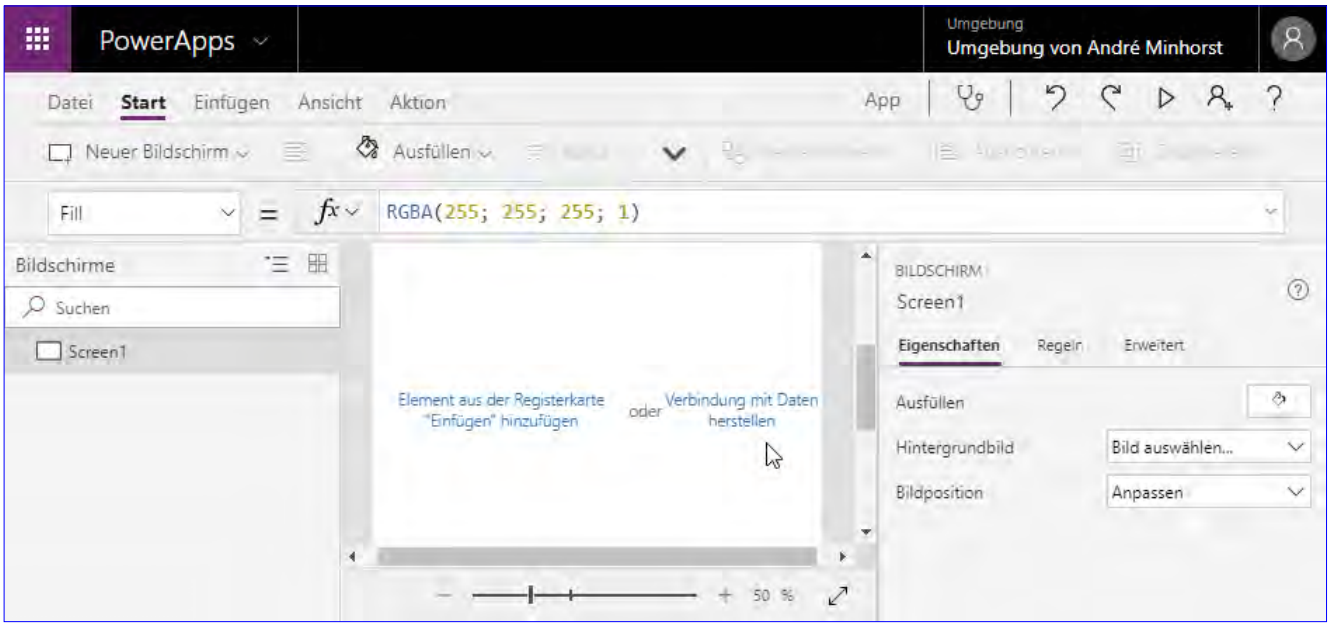


Bild 4: Entwicklungsumgebung für die PowerApps

verschiedener Datenquellen oder auch mit einer leeren App oder einer App-Vorlage. Es erscheint ein Bereich ähnlich dem Backstage-Bereich, den Sie von den Office-Anwendungen her kennen. Hier finden Sie auch einen Befehl namens **Neu** (siehe Bild 5). Hier klicken wir auf die Schaltfläche **Smartphonelayout** des Bereichs **App-Vorlagen**.

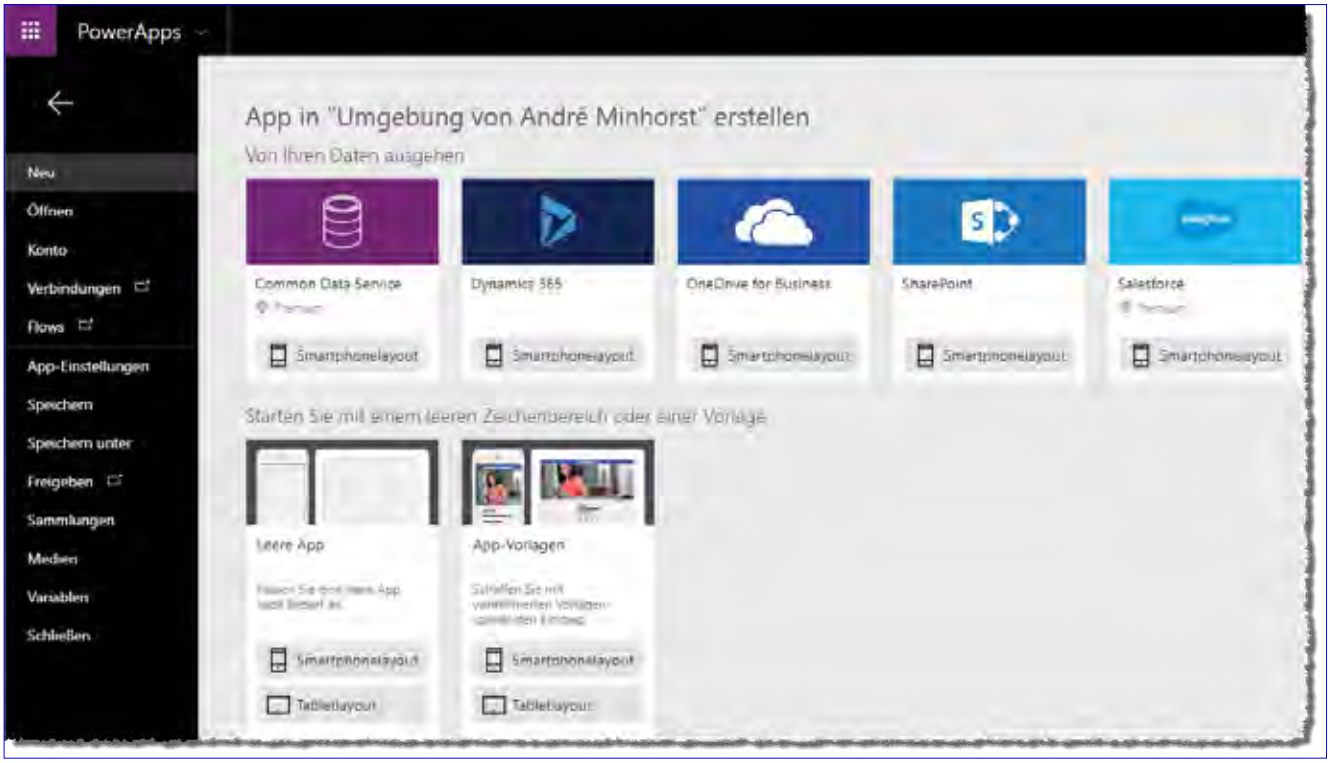


Bild 5: Backstage-Bereich mit dem **Neu**-Befehl

Vorlage auswählen

Wir entscheiden uns an dieser Stelle für den Eintrag **Budget Tracker** (siehe Bild 6). Außerdem müssen wir einen Speicherort für die Daten dieser App auswählen.

Klicken wir unten auf den Link **Auswählen**, erscheint ein Popup-Fenster mit Möglichkeiten wie **Dropbox**, **Google Drive** oder **OneDrive**.

Hier wälhen wir den Wert **Dropbox** und geben in einem weiteren Dialog den Zugriff auf die Dropbox frei.

Nach der Auswahl klicken wir auf die Schaltfläche **Verwendung**. Die Entwicklungsumgebung lädt anschließend die Vorlage, was einige Sekunden dauert.

Danach erscheint die erste Seite des Projekts, in diesem Fall die Seite **landing** (siehe Bild 7). Hier können Sie auf der linken Seite die einzelnen Elemente anklicken und diese mit den Eigenschaften auf der rechten Seite bearbeiten.

Bevor wir uns diese Features genauer ansehen, wollen wir die Anwendung einmal zum Laufen bringen.

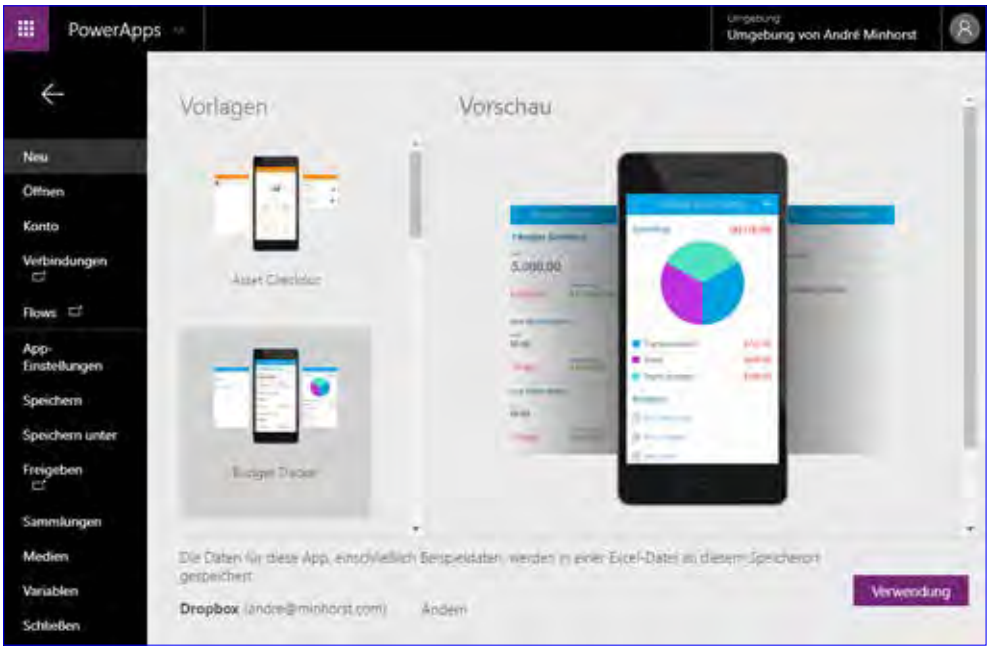


Bild 6: Auswahl der Vorlage

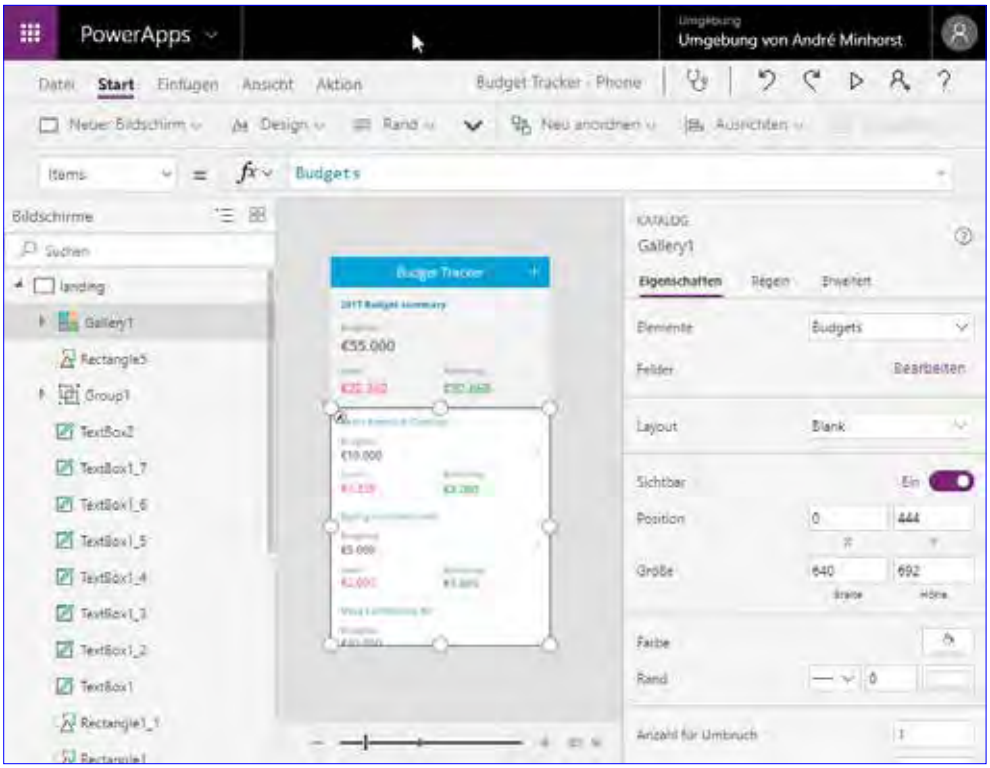


Bild 7: Bearbeiten der einzelnen Seiten der Vorlage

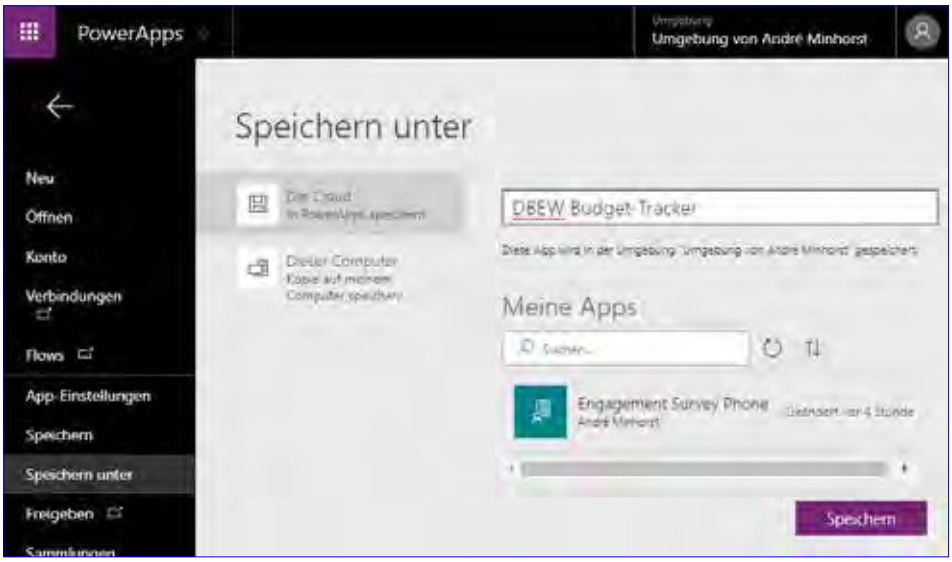


Bild 8: Speichern der PowerApp

Bild 9: Installieren der PowerApps-App

Anwendung freigeben

Um die Anwendung zur Benutzung freizugeben, speichern wir diese nun in der Cloud. Dazu klicken Sie wieder links oben auf **Datei** und wählen dann links den Befehl **Speichern** aus. Im nun erscheinenden Bereich geben Sie den Namen ein, unter dem die App gespeichert werden soll – in diesem Fall **DBEW Budget-Tracker** (siehe Bild 8). Dieser Vorgang dauert wiederum einige Sekunden. Anschließend erscheint die Bestätigung, dass die App gespeichert wurde. Hier finden Sie dann auch eine Schaltfläche namens **Diese App freigeben**. Betätigen Sie diese Schaltfläche, erscheint ein neuer Dialog in einem neuen Tab des Internet-Browsers, auf dem Sie die App für weitere Personen freigeben können. Das wollen wir in diesem Moment nicht tun – wir wollen die App nur über unseren eigenen Account nutzen.

Damit können wir diesen Schritt auslassen, die App ist nämlich bereits für den erstellenden Benutzer freigegeben.

PowerApps auf Smartphone verfügbar machen

Wenn Sie die PowerApp nun auf Ihrem Smartphone verfügbar machen wollen, benötigen

Bild 10: Anmelden mit dem Benutzeraccount

Bild 11: Auswahl des zu verwendenden Kontos bei mehrere Konten

PowerApp mit Datenbank erstellen

Im Artikel »Einstieg in PowerApps« haben wir uns angesehen, wie wir eine der Vorlagen in eine PowerApp umwandeln. Diese verwendet eine Excel-Datei als Datenquelle. Das ist schon ganz cool, aber wir wollen natürlich noch einen Schritt weitergehen – mit einer SQL Server-Datenbank als Datenquelle. Natürlich funktionieren auch noch eine ganze Reihe anderer Datenquellen, zum Beispiel MySQL. Wir schauen uns an, wie Sie eine PowerApp erstellen und dieser direkt die gewünschte Datenquelle hinzufügen.

Voraussetzungen

Voraussetzung für das Nachvollziehen der Beispiele dieses Artikels ist das Vorhandensein einer SQL Server-Instanz, die über das Internet erreichbar ist – zum Beispiel eine solche, wie wir sie im Artikel **SQL Server-Datenbank ins Web mit SQL Azure** erstellt haben. Die zweite Voraussetzung ist – wie auch schon für Azure – ein Microsoft-Konto und die Anmeldung zumindest an die Community-Version der PowerApps. Mehr darüber erfahren Sie im Artikel **Einstieg in PowerApps**.

Erstellen der PowerApp

Wenn Sie die oben genannten Voraussetzungen erfüllt haben, können Sie eine neue, leere PowerApp erstellen. Dazu öffnen Sie die Webseite **powerapps.microsoft.com**. Hier melden Sie sich über den **Anmelden**-Link an. Danach landen Sie in der Entwicklungsumgebung für PowerApps, die Sie ja auch schon im Artikel **Einstieg in PowerApps** kurz kennengelernt haben.

Nach dem Anmelden finden Sie die Startseite vor, auf der Sie verschiedene Vorlagen zum Anlegen neuer PowerApps vorfinden. Wir wählen die Vorlage **Canvas-App ohne Vorlage** aus, und zwar für Smartphones (siehe Bild 1).

Den nun erscheinenden Dialog **Willkommen bei PowerApps Studio** schließen wir mit der Schaltfläche **Überspringen**. Danach finden Sie das PowerApps Studio wie in Bild 2 vor.

Verbindung mit Daten herstellen

Die Entwicklungsumgebung zeigt im Arbeitsbereich die beiden Möglichkeiten **Elemente aus der Registerseite "Einfügen" hinzufügen** und **Verbindung mit Daten her-**

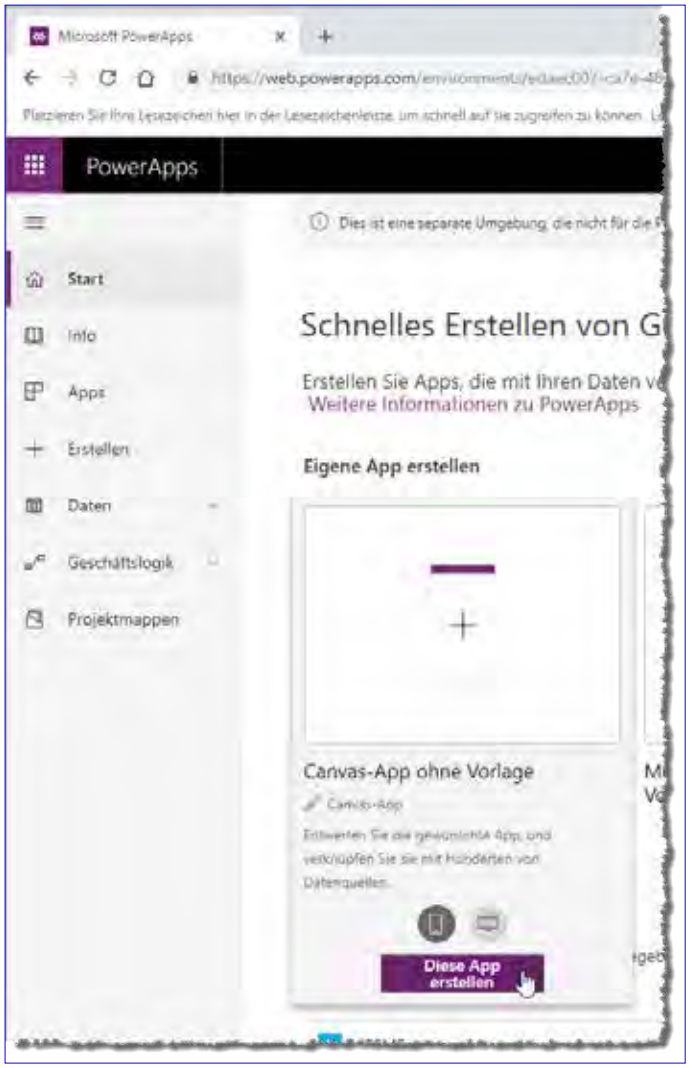


Bild 1: Erste Ansicht der Entwicklungsumgebung

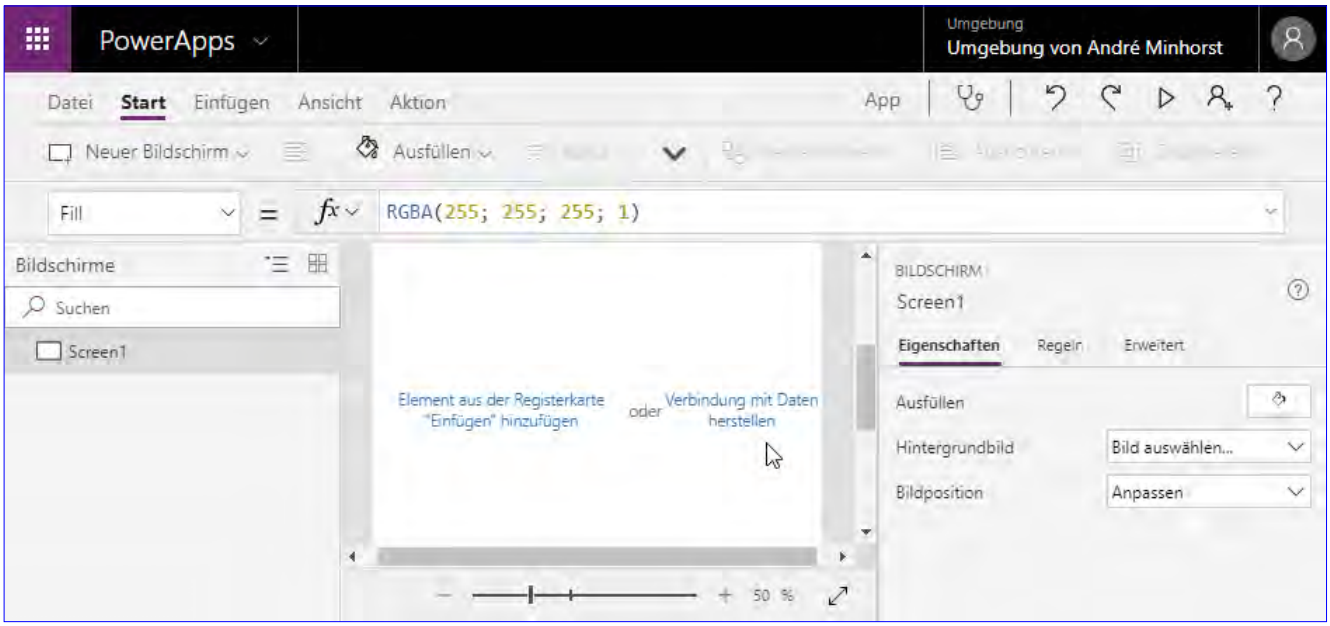


Bild 2: PowerApps-Studio

stellen an. Wir wollen eine Verbindung mit Daten herstellen, also klicken wir auf **Verbindung mit Daten herstellen**.

Hier bietet ein kleiner Dialog namens **Daten** die Möglichkeit, statische Daten aus Excel zu importieren (siehe Bild 3). Wir wollen aber direkt einen Schritt weitergehen und die Daten aus einer SQL Server-Tabelle einer Azure-Datenbank laden.

Also klicken wir auf **Neue Verbindung** und finden dann eine sehr lange Liste verschiedenster Datenquellen vor. Uns interessiert jedoch der SQL Server, den wir am schnellsten finden, indem wir **SQL** oben ins Suchfeld eingeben. Es erscheinen dann die Einträge aus Bild 4.

Wie Sie eine Azure-Datenbank anlegen, haben wir ja im Artikel **SQL Server-Datenbank ins Web mit SQL Azure** schon beschrieben. Dort sollten Sie sich einige Daten gemerkt haben wie die Adresse des SQL Servers, den Datenbanknamen, den Namen des Benutzers sowie das Kennwort des Benutzers.

Diese können Sie nun im folgenden Dialog eingeben, der genau diese Informationen abfragt (siehe Bild 5). Hier finden Sie auch noch die Option **Verbindung mithilfe eines lokalen Datengateways**. Sie können damit nach einigen weiteren Maßnahmen auch auf einen SQL Server zugreifen, der auf ihrem lokalen Rechner liegt, aber von außen zugreifbar sein

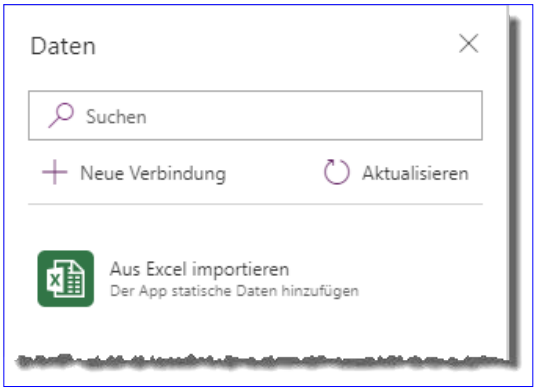


Bild 3: Auswahl der Datenquelle

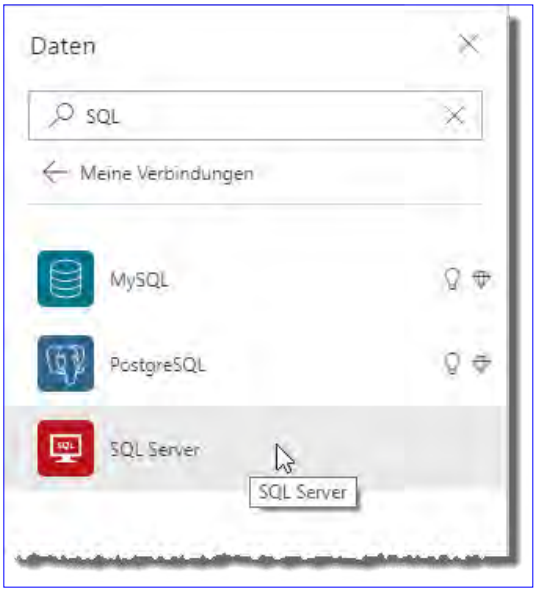


Bild 4: Filtern der Datenquellen

PowerApps: Artikel verwalten

In den vorherigen Artikeln haben wir uns schon eine PowerApp anhand einer Vorlage angesehen und eine kleine, an die Daten einer Tabelle gebundene PowerApp programmiert. Nun gehen wir einen Schritt weiter und erstellen eine PowerApp mit den für die Anzeige von Daten wichtigen Steuerelementen und zeigen, wie Sie die Steuerelemente mit Daten füllen, zwischen verschiedenen Seiten navigieren und Daten anlegen, bearbeiten oder auch löschen.

Voraussetzungen

Um die Beispiele dieses Artikels nachvollziehen zu können, benötigen Sie eine SQL Server-, MySQL- oder PostgreSQL-Datenbank, die über das Internet erreichbar ist. Sie können dazu eine Datenbank von einem Webserver nehmen, den Sie selbst betreiben oder auch einen Microsoft Azure-Account mit einer SQL Server-Datenbank anlegen (siehe auch im Artikel **SQL Server-Datenbank ins Web mit SQL Azure**).

Außerdem setzen wir auf der PowerApp auf, die wir im Artikel **PowerApp mit Datenbank erstellen angelegt** haben. Dort haben wir der PowerApp zum Beispiel einige Tabellen der Datenbank **Suedsturm_SQL** eines SQL Servers hinzugefügt. Den Start machen wir über die Webseite **powerapps.microsoft.com**, wo Sie sich mit Ihren Zugangsdaten einloggen und die gewünschte Anwendung aufrufen oder neu anlegen.

Bild 1: Einfacher Link

Liste erstellen

Wir beginnen mit einer ersten Seite, die eine Liste von Elementen einer unserer Tabellen enthalten soll. Dazu wählen wie in der Entwicklungsumgebung unter **Einfügen** den Eintrag **Neuer Bildschirm\|Liste** aus (siehe Bild 1).

Danach zeigt die Entwicklungsumgebung, die wir uns hier nochmal genauer anschauen, die neue Seite direkt mit den von der Vorlage vorgesehenen Elementen an (siehe Bild 2). Diese sehen wir im mittleren Bereich. Die einzelnen Elemente dieser Seite finden wir im Bereich links unter **Screen2**. Das Element **BrowseGallery1** ist das eigentliche Element zur Anzeige mehrerer Datensätze der Datenquelle, das im Prinzip wie eine Art Endlosformular wie unter Access arbeitet. Die Anzeige im Entwurf ist etwas irreführend, denn eigentlich enthält das Steuerelement jedes der abgebildeten untergeordneten Steuerelemente nur einmal. Hier werden allerdings direkt Platzhalterdaten abgebildet.

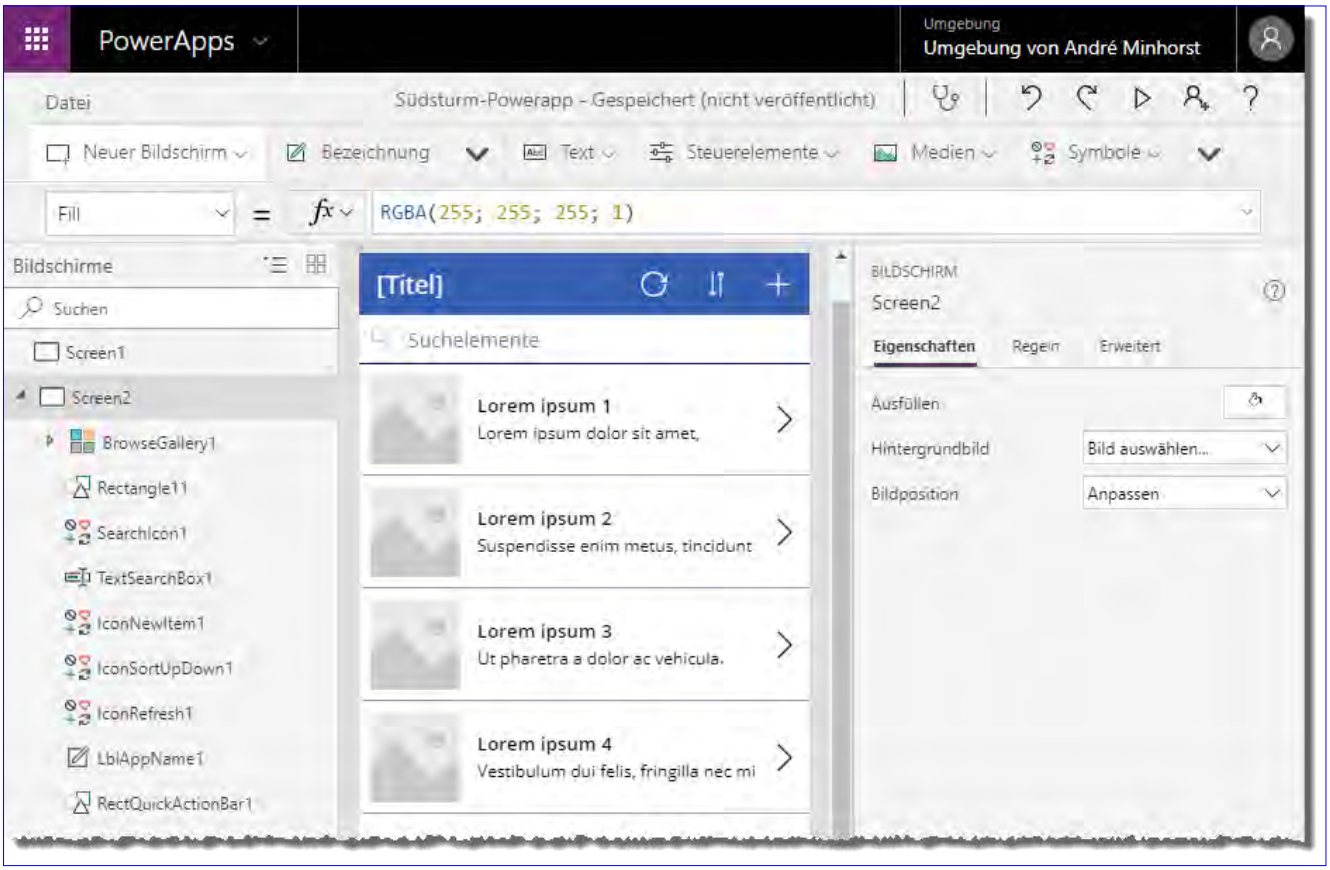


Bild 2: Neue Seite in der Entwicklungsumgebung

Wenn Sie auf der linken Seite **BrowseGallery1** aufklappen, finden Sie die für ein Element dieser Endlosansicht angelegten Steuerelemente (siehe Bild 3) – einen Separator, ein Image, eine Schaltfläche sowie zwei Bezeichnungsfelder für Texte aus der Datenquelle.

Hier können wir uns schon einiges abschauen, aber wir wollen mit einer komplett leeren Seite beginnen. Dazu fügen wir eine neue Seite hinzu, die wir diesmal auf Basis der Formularvorlage **Leer** erstellen. Dementsprechend ist die Seite auch komplett leer. Wir wollen eine Übersicht aller Artikelnamen anzeigen, für die es oben ein Textfeld für eine Schnellsuche gibt. Dieses

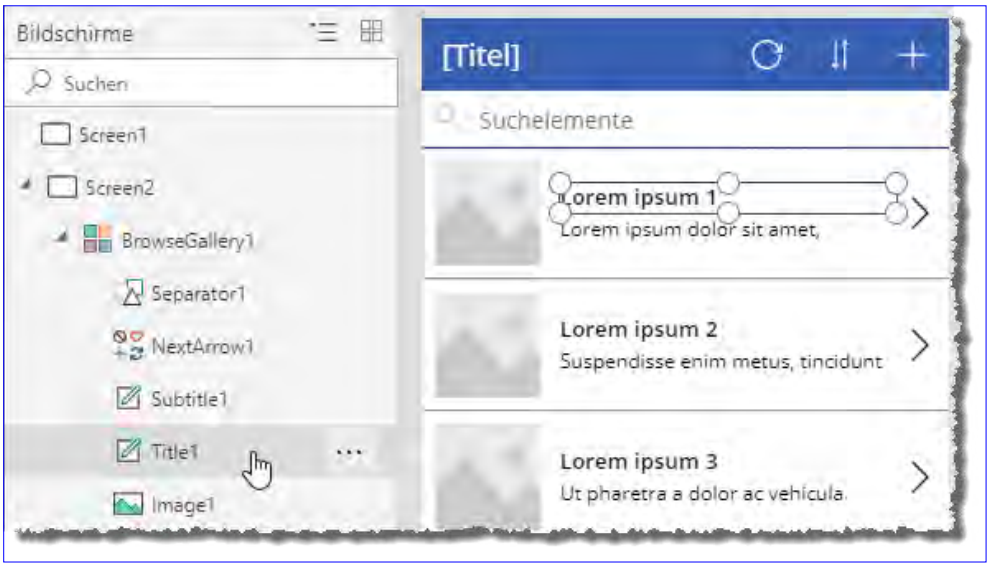


Bild 3: Untersuchen der einzelnen Elemente der Gallery

soll nach der Eingabe eines jeden Zeichens das Suchergebnis filtern.

Dazu fügen wir der neuen Seite, die wir zuvor in **Artikel-uebersicht** umbenennen, als Erstes ein leeres Katalog-Element hinzu, das wir im Bereich **Einfügen** unter **Katalog|Leerzeichen vertikal** vorfinden (siehe Bild 4).

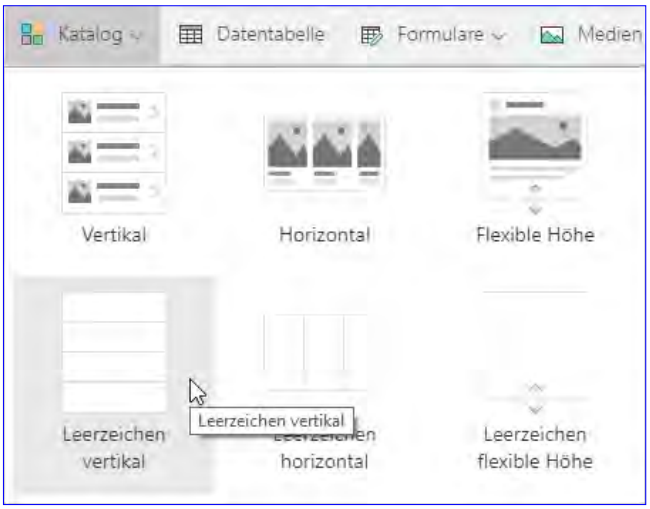


Bild 4: Einfügen eines leeren Katalogs mit vertikalen Elementen

Danach legen wir noch ein Bezeichnungsfeld namens **lblSuche** und ein Textfeld namens **txtSuche** über dem Katalog-Element namens **galArtikel** an (siehe Bild 5).

Datenquelle für den Katalog hinzufügen

Die Datenquelle fügen wir nun hinzu, indem wir das Katalog-Element markieren und dann rechts in den Eigenschaften die gewünschte Tabelle auswählen – in diesem Fall **tblArtikel** (siehe Bild 6). Dies legt auch noch den Wert der Eigenschaft **Items** im Bereich **Daten** der Registerseite **Erweitert** auf **'[dbo].[tblArtikel]'** fest.

Dann fügen wir bei markiertem Katalog-Element ein weiteres Bezeichnungsfeld hinzu, dass dann in das Katalog-Element eingefügt wird. Das Bezeichnungsfeld kann – im Gegensatz zu Access – auch an eine Datenherkunft gebunden werden und dient so etwa zur Anzeige von gebundenen Werten, die aktuell nicht vom Benutzer bearbeitet werden sollen. Dazu würden wir dann das Textfeld nutzen. Es geschieht etwas Überraschendes: Sobald wir im Bereich **Einfügen** auf das Element **Bezeichnung** geklickt haben, um dieses hinzuzufügen, enthält dieses bereits die Werte des Feldes **Artikelname** der Tabelle **tblArtikel**!

Bild 5: Steuerelemente für die Artikelübersicht

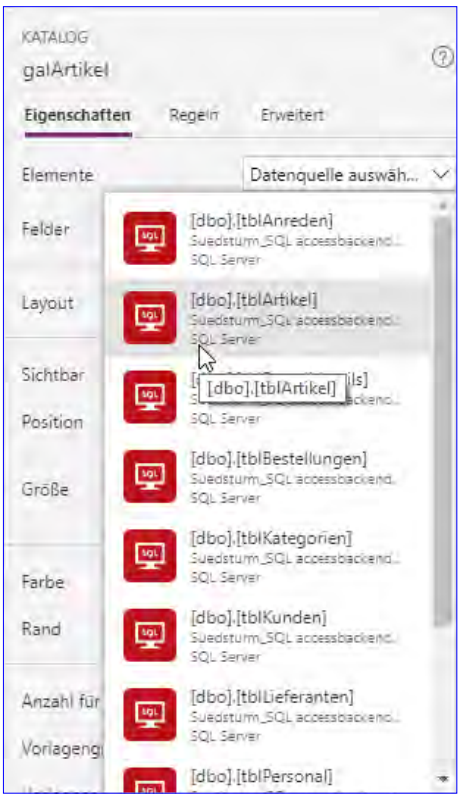


Bild 6: Auswahl der Datenquelle

Diese Anzeige sieht, nachdem wir die Höhe des obersten Bereichs des Katalog-Elements etwas verkleinert haben, etwa wie in Bild 7 aus. Offensichtlich hat

die Entwicklungsumgebung das erste Textfeld der Datenquelle ermittelt und dieses als Datenherkunft für das Bezeichnungsfeld eingestellt. Schauen wir also nach, welche Eigenschaft mit einem entsprechenden Wert ausgestattet wurde.

Die passende Eigenschaft finden wir dann auf der Seite **Erweitert** der Eigenschaften des Bezeichnungsfeldes. Die Bindung erfolgt über die Eigenschaft **Text**, der automatisch der Wert **ThisItem.Artikelname** zugewiesen wurde (siehe Bild 8). **ThisItem** referenziert dabei offensichtlich die für die Eigenschaft **Elemente** des Katalog-Steuerelements angegebene Datenquelle.

Bild 7: Anzeige der Daten in der Vorschau

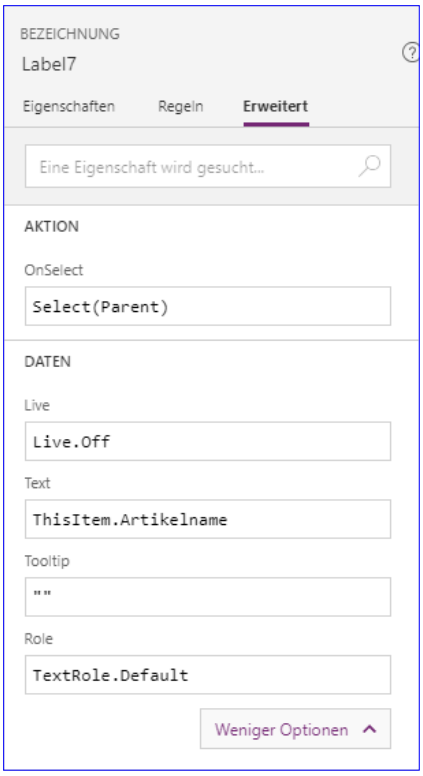


Bild 8: Eigenschaft zum Einstellen der Datenherkunft eines Steuerelements

Suche hinzufügen

Nun wollen wir das Such-Textfeld in die Anzeige der Daten im Katalog-Steuerelement einbinden. Dazu müssen wir den Wert der Eigenschaft **Items** des Katalog-Steuerelements anpassen. Bisher hat diese Eigenschaft den Wert **'[dbo].[tblArtikel]'**.

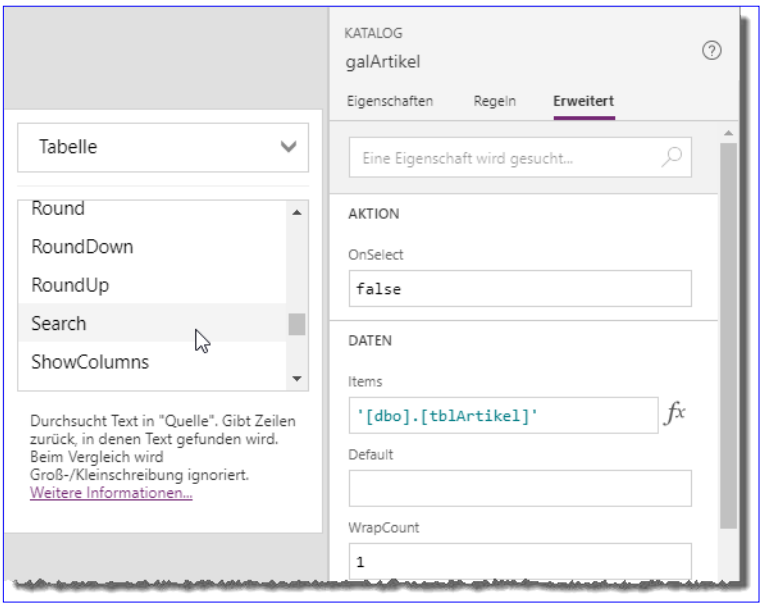


Bild 9: Auswählen einer Funktion zum Suchen von Daten

Nun haben PowerApps einen speziellen Dialog von Ausdrücken und Funktionen, die am ehesten mit denen von Excel zu vergleichen sind. Diese können Sie einfach auswählen, wenn Sie die Eigenschaft **Items** markieren und dann auf die rechts davon erscheinende Schaltfläche mit dem Funktions-Icon klicken. Es erscheint dann links daneben ein Bereich zur Auswahl der zur Verfügung stehenden Funktionen. Hier wählen wir oben den Eintrag **Tabelle** aus und scrollen dann zur Funktion **Search** (siehe Bild 9).

Klicken wir doppelt auf diesen Eintrag, wird der erste Teil der Funktion schon einmal in die Eigenschaft **Items** eingetragen (siehe Bild 10). Außerdem erscheint direkt ein Popup mit Vorschlä-

gen für den ersten Parameter, für den wir die Tabelle **dbo.tblArtikel** auswählen. Die **Search**-Funktion erwartet mindestens drei Parameter:

- Name der Datenquelle, hier **'[dbo].[tblArtikel]'**
- Steuerelement, das den Vergleichswert liefert. Dabei muss auch die Eigenschaft angegeben werden, die den Wert liefert, in diesem Fall also etwa **txtSuche.Text**.
- Schließlich folgen ein oder mehrere Felder der Tabelle, in denen nach dem mit dem zweiten Parameter angegebenen Wert gesucht werden soll.

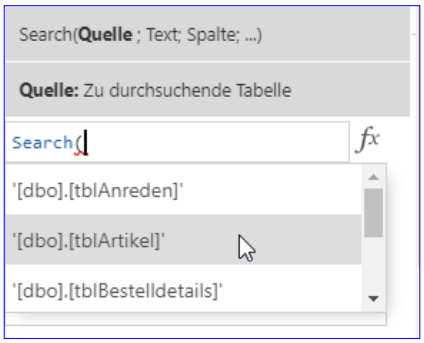


Bild 10: Eingabe der Funktionsparameter

Unser Suchausdruck sieht nun wie folgt aus (siehe Bild 11):

```
Search('[dbo].[tblArtikel]';txtSuche.Text;"Artikelname")
```

Bild 11: Leeres Suchergebnis

Warum aber sind nach der Eingabe dieses Ausdrucks plötzlich alle Einträge des Katalog-Elements leer? Die Lösung ist einfach: Für das Textfeld ist der Standardwert **Texteingabe** festgelegt. Wenn Sie doppelt in das Textfeld klicken und dieses leeren, erscheinen auch alle Einträge der Tabelle **tblArtikel** wieder (siehe Bild 12).

Wenn wir nun einen Suchbegriff eingeben, wird die Liste der Artikel auch direkt gefiltert (siehe Bild 13).

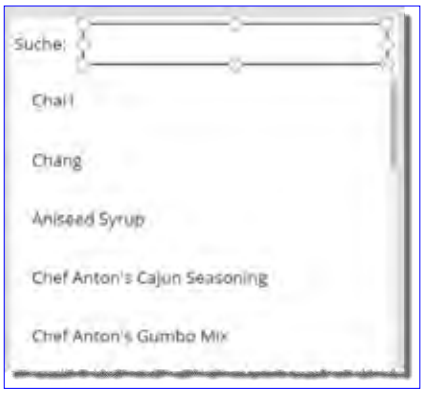


Bild 12: Leeres Suchfeld

Damit demnächst immer direkt alle Einträge angezeigt werden, leeren Sie die Eigenschaft **Default** des Steuerelements **txtSuche** am besten noch. Das funktioniert allerdings nicht durch vollständiges Leeren, sondern Sie müssen eine leere Zeichenkette (**""**) für diese Eigenschaft eintragen.

Suchfunktion auf dem Smartphone testen

Das wollen wir nun auf dem Smartphone ausprobieren. Auf diesem muss dazu die App **PowerApps** installiert sein, außerdem melden Sie sich unter den gleichen Daten an, unter denen Sie auch die PowerApp erstellt haben.

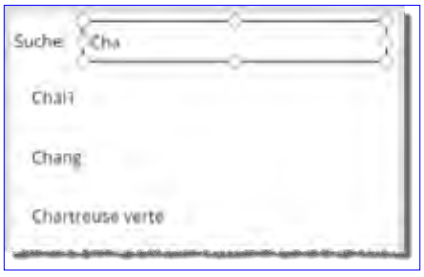


Bild 13: Suchergebnisse

Bevor wir zum Smartphone wechseln, müssen wir allerdings noch folgende Schritte durchführen:

- Unter Umständen ist die neu erstellte Seite nicht die erste Seite in der Liste der Seiten. In diesem Fall bewegen Sie diese mit dem Kontextmenü-Befehl **Nach oben** an die erste Position.
- Danach müssen wir die App noch speichern. Das gelingt ganz einfach mit dem Befehl **DateiSpeichern**.

- Schließlich veröffentlichen Sie die PowerApp noch mit der Schaltfläche **Veröffentlichen**, die nach dem Speichern im gleichen Bereich erscheint.

Danach können Sie die Anwendung in der PowerApps-App auf Ihrem Smartphone öffnen – und sie funktioniert wie gewünscht! Ich finde das ein sehr bemerkenswertes Ergebnis gemessen daran, dass wir – wenn die Schritte etwas eingeübt sind – wohl nur wenige Minuten für die Erstellung dieser App benötigen werden.

Wenn Sie kein Smartphone zur Hand haben, können Sie die App übrigens auch direkt auf dem Rechner ausprobieren. Dazu klicken Sie auf der Übersichtsseite für die PowerApps einfach auf den Eintrag mit dem Namen der zu testenden App.

Details anzeigen

Als nächstes wollen wir der Katalog-Seite Elemente hinzufügen, die es dem Benutzer ermöglichen, die Details zu jedem der angezeigten Datensätze zu öffnen. Dazu brauchen wir natürlich nicht nur eine entsprechende Schaltfläche je Eintrag, sondern auch noch eine Seite zur Anzeige der Details. Diese wollen wir zuerst erstellen.

Dazu fügen wir über den Befehl **Einfügen|Neuer Bildschirm** einen weiteren neuen Bildschirm hinzu, wieder mit der Vorlage **Leer**. Diese Seite nennen wir **Artikeldetails**.

Dann fügen wir über den Menüeintrag **Formulare|Anzeige** ein Formular zur Seite hinzu, das die Anzeige der Details zum gewählten Artikel erlauben soll. Auch dieses zeigt wieder an, dass es noch mit Daten verbunden werden muss. Also wählen wir als Datenquelle wieder die Tabelle **[dbo].[tblArtikel]** aus (siehe Bild 14) und klicken auf die Schaltfläche **Verbinden**.

Danach wird es richtig komfortabel: Für die Eigenschaft **Felder** finden Sie einen Link namens **Felder bearbeiten** (siehe Bild 15). Klicken Sie auf diesen Link, erscheint links davon ein neuer Bereich namens **Felder**. Hier klicken Sie auf den Link **Feld hinzufügen**, um eine Liste aller Felder der unter **Datenquelle** angegebenen Tabelle anzuzeigen. Sie können diese dann bequem anhaken.

Klicken Sie dann auf die Schaltfläche **Hinzufügen**, fügt die Anwendung für jedes Feld ein paar Elemente zum Entwurf hinzu. Dass es sich um mehrere Elemente handelt, können Sie am besten an der Liste der Steuerelemente auf der rechten Seite erkennen (siehe Bild 16). Hier erscheint für jedes Feld ein Steuerelement des Typs **Karte**, das jeweils zwei Elemente des Typs

Bild 14: Hinzufügen der Daten zum Formular

Bezeichnung enthält. Das erste Bezeichnungs-Steuerelement zeigt dabei den Feldnamen an, das zweite den Inhalt des Feldes.

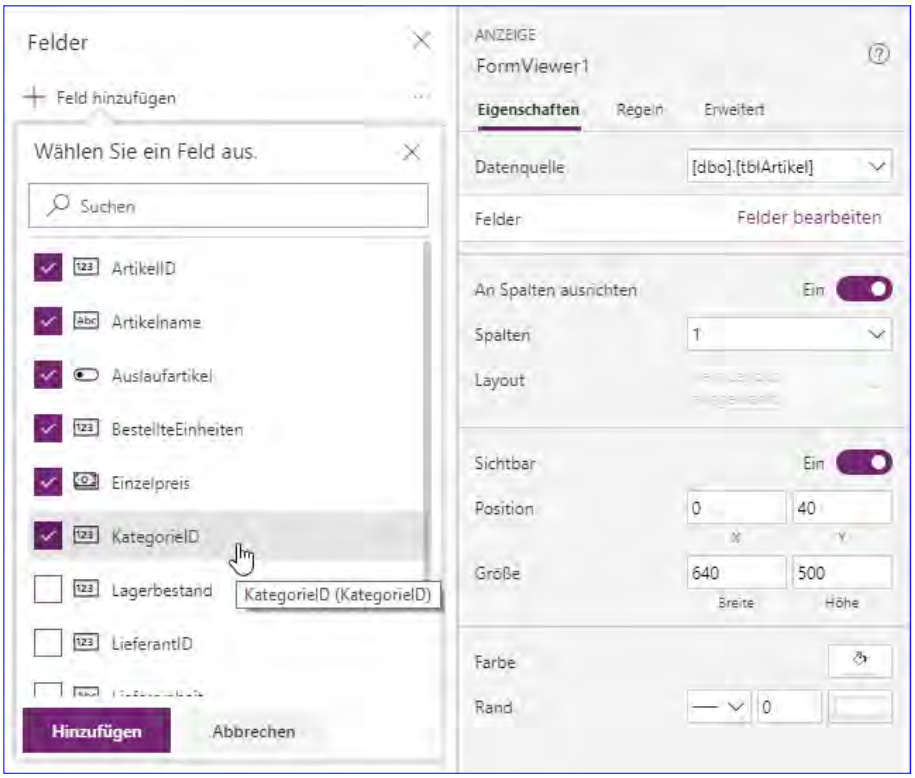


Bild 15: Auswählen der Felder

Damit die Felder wissen, was sie anzeigen müssen, enthält das **Karte**-Element ein paar passende Eigenschaften (siehe Bild 17):

- **DataField**: Feld der Tabelle, an welches das Steuerelement gebunden ist
- **DisplayName**: Anzuzeigender Text des Bezeichnungsfeldes
- **Required**: Gibt an, ob die Eingabe in das Feld erforderlich ist.
- **Default**: Standardwert, der aus dem tatsächlichen Wert des Feldes für den angezeigten Datensatz gewonnen wird

Nun soll die Detailansicht eines Artikels natürlich auch einen Artikel anzeigen, was aber noch nicht geschieht – aktuell erscheinen nur die Bezeichnungsfelder mit den Feldnamen.

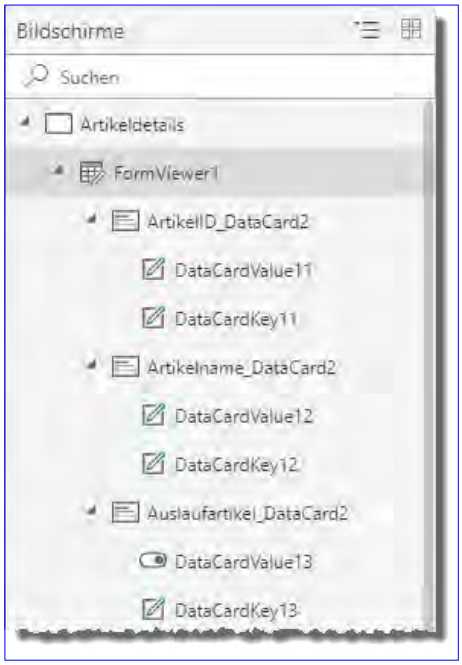


Bild 16: Steuerelemente zur Anzeige von Daten

Um Daten anzuzeigen, müssen wir dem Detailformular noch mitteilen, welchen Datensatz der Datenquelle es anzeigen soll. Dies stellen wir mit der Eigenschaft **Item** ein, die wir im **Eigenschaften**-Bereich **Erweitert** des **Anzeige**-Elements finden. Hier tragen wir den Wert **First('[dbo].[tblArtikel]')** ein (siehe Bild 18). Der Ausdruck

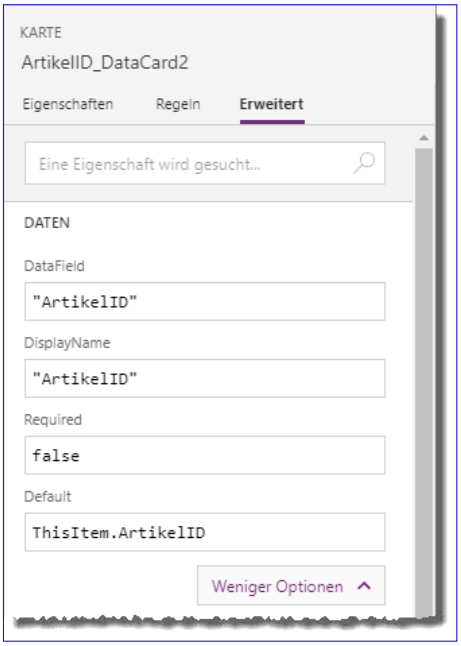


Bild 17: Eigenschaften dieser Steuerelemente

Powerapps: Datensatz per DropDown auswählen

Eine oft verwendete Darstellung ist die, bei der sich oben im Fenster ein Kombinationsfeld befindet, mit dem der Benutzer dann den Datensatz auswählt, dessen Detaildaten in den übrigen Steuerelementen im unteren Bereich angezeigt werden sollen. Dieser Artikel zeigt, aufbauend auf den vorherigen Artikeln, wie Sie einen Bildschirm erzeugen, der im oberen Bereich ein solches Kombinationsfeld liefert und im unteren Bereich die Daten des gewählten Datensatzes anzeigt. Die Daten stammen dabei aus der Kunden-Tabelle der Beispieldatenbank Suedsturm_SQL.

Wir setzen auf unserem PowerApps-Projekt auf, das wir im vorherigen Artikel bereits begonnen haben. Wir verwenden also die Datenbank **Suedsturm_SQL** als Datenquelle.

Um nicht immer wieder die neu hinzufügten Bildschirmseiten nach oben im Projekt verschieben zu müssen, um diese direkt beim Öffnen der PowerApp testen zu können, haben wir nun einen Startbildschirm erstellt. Dieser sieht wie in Bild 1 aus. Für die Schaltfläche **Kundenauswahl** haben wir für die Eigenschaft **OnSelect** den folgenden Ausdruck festgelegt:

```
Navigate(Kundenauswahl;
ScreenTransition.Fade)
```

Damit öffnen wir per Mausklick auf die Schaltfläche einen weiteren Bildschirm namens **Kundenauswahl**. Den fügen wir mit dem Befehl **Neuer Bildschirm\|Leer** zum Projekt hinzu.

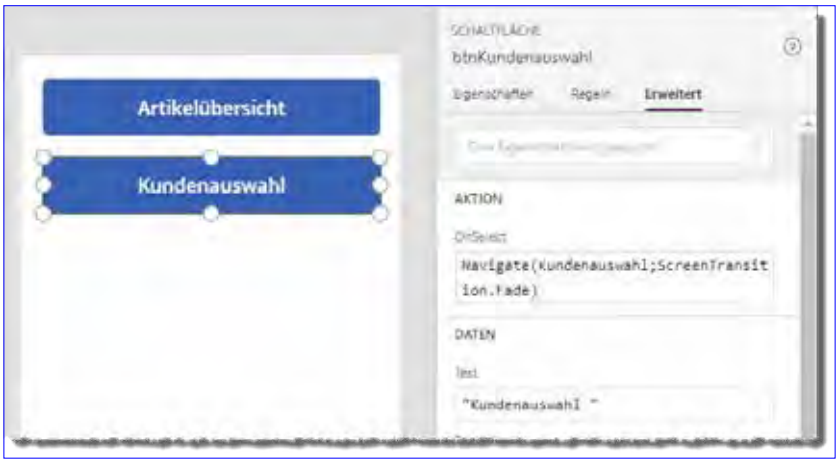


Bild 1: Hinzufügen einer Schaltfläche zum Öffnen einer weiteren Bildschirmseite

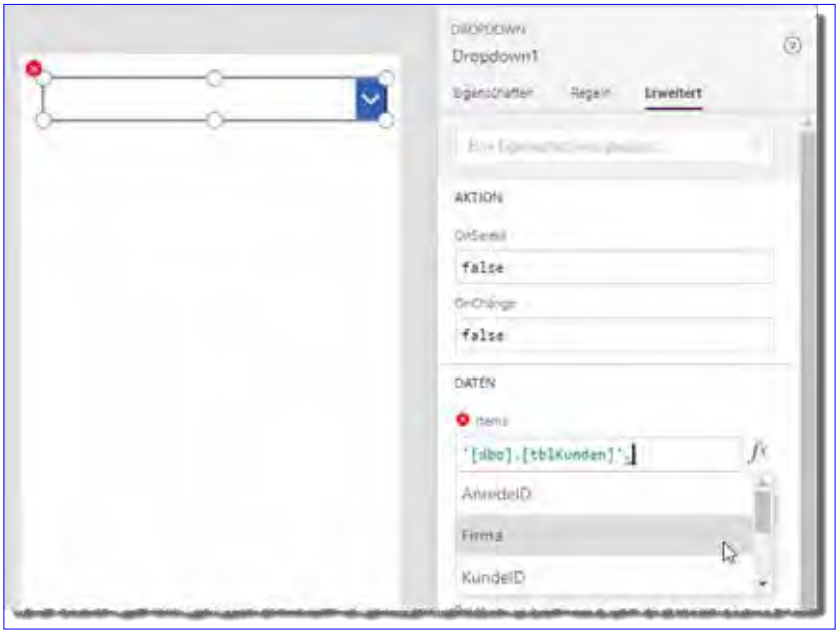


Bild 2: Anlegen des Dropdown-Steuerelements

Dropdown hinzufügen

Für die Auswahl der anzuzeigenden Daten verwenden wir das **Dropdown**-Steuerelement, das wir mit dem Befehl **Steuerelemente\|Dropdown** zur Bild-

schirmseite **Kundenauswahl** hinzufügen. Damit dieses beispielsweise die Werte des Feldes Firma der Tabelle **tblKunden** anzeigt, tragen Sie den Wert **'[dbo].[tblKunden]'** in die Eigenschaft **Items** ein und fügen noch einen Punkt hinzu.

Danach bietet die Benutzeroberfläche eine Liste der enthaltenen Felder an, die Sie so einfach auswählen und festlegen können (siehe Bild 2). Außerdem ändern wir noch den Namen des Dropdown-Steuerelements in **drpKunden**.

Interessanterweise werden die Daten hier nicht direkt zur Entwurfszeit angezeigt, wie es in den anderen Artikeln schon der Fall war. Also probieren wir das Dropdown-Feld im Smartphone aus oder im Webplayer. Dazu speichern wir die aktuelle Version der App mit dem Befehl **DateiSpeichern**.

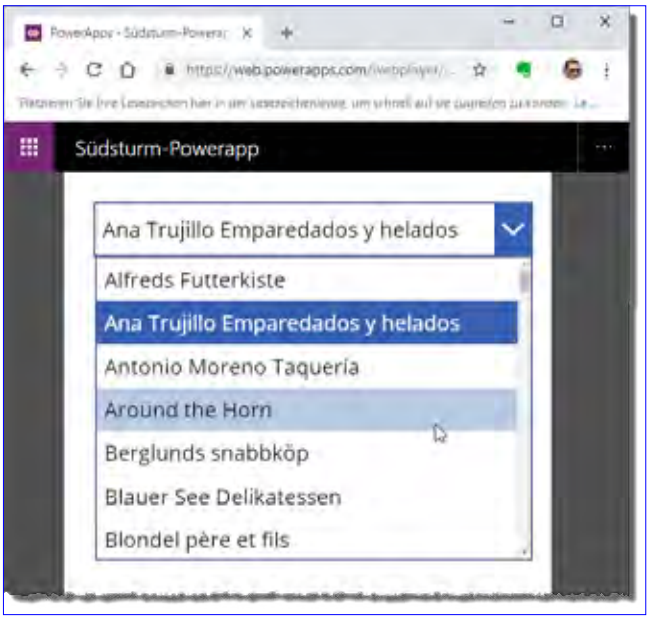


Bild 3: Testen der App im Webplayer

Anschließend müssen Sie diese über den im gleichen Bereich erscheinenden Befehl **Veröffentlichen** noch publizieren. Danach liefert diese Bildschirmseite im Webplayer die Ansicht aus Bild 3.

Bereich zur Anzeige der Daten hinzufügen

Nun benötigen wir unterhalb des Dropdown-Elements weitere Steuerelemente, um die Felder des gewählten Datensatzes anzuzeigen. Dazu fügen wir der Bildschirmseite zunächst ein übergeordnetes Element hinzu, und zwar über den Befehl **FormularelBearbeiten** des Bereichs **Einfügen**.

Das so hinzugefügte Element platzieren wir so auf der Bildschirmseite, dass es sich unterhalb des Dropdown-Steuerelements befindet und den verbleibenden Platz einnimmt (siehe Bild 4). Außerdem stellen wir seinen Namen auf **frmKundenuebersicht** ein.

Bild 4: Das neue Formular ist noch nicht mit Daten verbunden.

Dann wählen Sie für die Eigenschaft **Datenquelle** im Bereich **Eigenschaften** den Wert **[dbo].[tblKunden]** aus. Dies stellt automatisch die Eigenschaft **DataSource** im Bereich **Erweitert** auf den Wert **'[dbo].[tblKunden]'** ein.

Darunter finden Sie die Eigenschaft **Item**, der Sie folgenden Ausdruck zuweisen:

```
First(Filter('[dbo].[tblKunden]';Firma=drpKunden.Selected.Value))
```

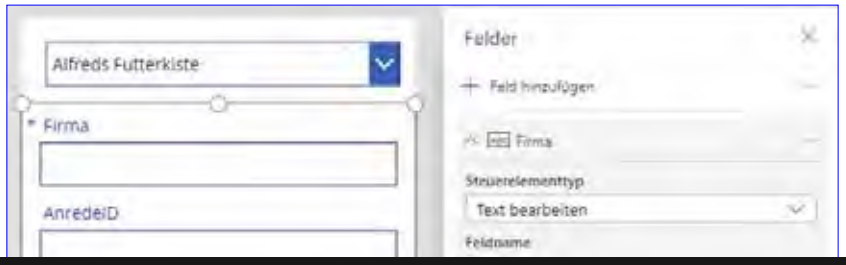
Mittlerweile hat sich auch der Inhalt des Formulars zum Bearbeiten der Daten geändert. Dieses zeigt nun den Text aus Bild 5 an, der darauf hinweist, dass noch die Felder für Anzeige ausgewählt werden müssen.

Bild 5: Es fehlen noch die Felder zum Anzeigen der Daten.

Klicken wir auf diesen Link, erscheint der Bereich **Felder**, in dem wir mit einem Klick auf **Feld hinzufügen** eine Liste der Felder der Datenherkunft anzeigen (siehe Bild 6).

Hier wählen wir nun die gewünschten Felder aus und klicken auf **Hinzufügen**. Wenn Sie die Felder in einer bestimmten Reihenfolge hinzufügen wollen, können Sie das nacheinander erledigen, indem Sie zunächst den obersten Eintrag auswählen und mit **Hinzufügen** im Formular platzieren, dann wieder auf **Feld hinzufügen** klicken und das nächste Feld hinzufügen. Wenn Sie zwischendurch den Dialog **Felder** schließen, können Sie diesen wieder öffnen, indem Sie im Eigenschaftenblatt auf den Link **Felder bearbeiten** rechts neben der Eigenschaft **Felder** klicken (siehe Bild 7).

Auf diese Weise fügen wir ein Feld nach dem anderen zum Formular hinzu. Sie können allerdings auch die Felder, die Sie hinzufügen wollen, einfach in der richtigen Reihenfolge in der Feldliste anklicken – diese werden dann in dieser Reihenfolge zum Formular hinzugefügt.



Die Leseprobe dieses Artikels ist hier zu Ende.

Willst Du mehr?

ZUM SHOP

2.400 Seiten Know-how rund um die Datenbankentwicklung mit .NET

André Minhorst Verlag

Spare 20 EUR mit Code dbew20

AUSGABE 01/2019

DATENBANK ENTWICKLER

MAGAZIN FÜR DIE DATENBANKENTWICKLUNG MIT VISUAL STUDIO FÜR DESKTOP, WEB UND CO.

TOP-THEMEN:

POWERAPPS	Kamera steuern	SEITE 3
POWERAPPS	Bilder in Datenbank speichern	SEITE 8
ACCESS ZU .NET	Von Access zu Entity Framework: Update 1	SEITE 15
ACCESS ZU WPF	Detailformulare mit Textfeldern	SEITE 25
EF	Daten abfragen mit VB und LINQ	SEITE 38

André Minhorst Verlag

PowerApps: Kamera steuern

PowerApps sind die Lösung von Microsoft, um schnell Anwendungen für Smartphones und Co zu realisieren. Und da es kein Smartphone ohne Kamera gibt, stellt sich natürlich die Frage, ob wir PowerApps auch so programmieren können, dass wir damit Fotos aufnehmen und diese weiterverarbeiten können – etwa, indem wir diese in einer Datenbank oder in der Cloud speichern oder sogar in den Bildern des Smartphones. Der vorliegende Artikel zeigt, wie Sie eine Funktion zum Fotografieren zu Ihrer PowerApp hinzufügen und was Sie mit den aufgenommenen Bildern alles tun können.

Neue PowerApp anlegen

Die Funktion zum Aufnehmen von Fotos und zum Weiterverarbeiten der Aufnahmen wollen wir in einer neuen PowerApp ausprobieren, daher legen Sie unter **web.powerapps.com** nach der Anmeldung mit Ihren Kontodaten mit einem Klick auf Erstellen und der nachfolgenden Auswahl von **Canvas-App ohne Vorlage** eine neue App an (siehe Bild 1).

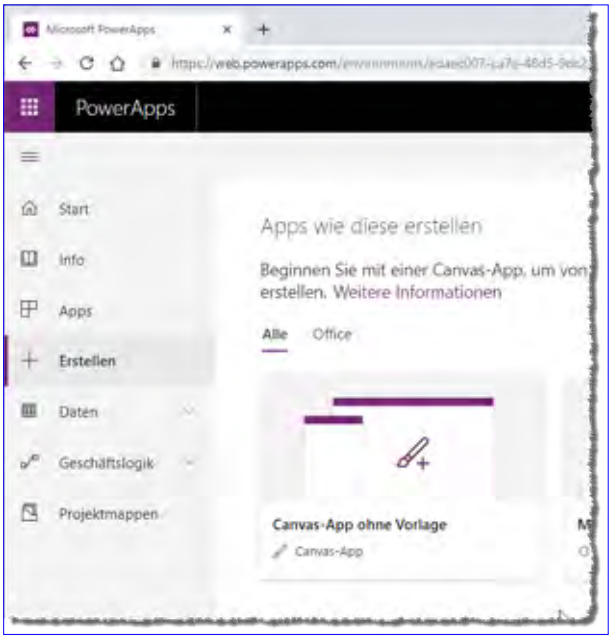


Bild 1: Anlegen der neuen PowerApp

Im folgenden Dialog geben wir als Namen für die App schlicht **Fotos** an und legen fest, dass wir eine App für das Telefon erstellen wollen (siehe Bild 2).

Kamera-Element hinzufügen

Nun empfängt uns die App mit einer neuen, leeren Seite. Hier wollen wir als Erstes ein **Kamera**-Element einfügen. Dazu wählen Sie ganz oben Einfügen aus und aus der nun erscheinenden unteren Menüleiste den Eintrag **MedienlKamera** (siehe Bild 3).

Diese erscheint dann als schwarzer Bereich mit dem Text **Ihre Kamera ist nicht eingerichtet, oder Sie verwenden sie bereits**. Diese Meldung erscheint beispielsweise dann, wenn Sie an einem Rechner mit einem Monitor arbeiten, an dem keine Kamera zur Verfügung steht. Das macht das Testen natürlich etwas komplizierter, aber Sie können dann auch direkt am Smartphone testen. Dazu speichern Sie die Anwendung einfach mit dem Menübefehl **DateilSpeichern** und betätigen danach

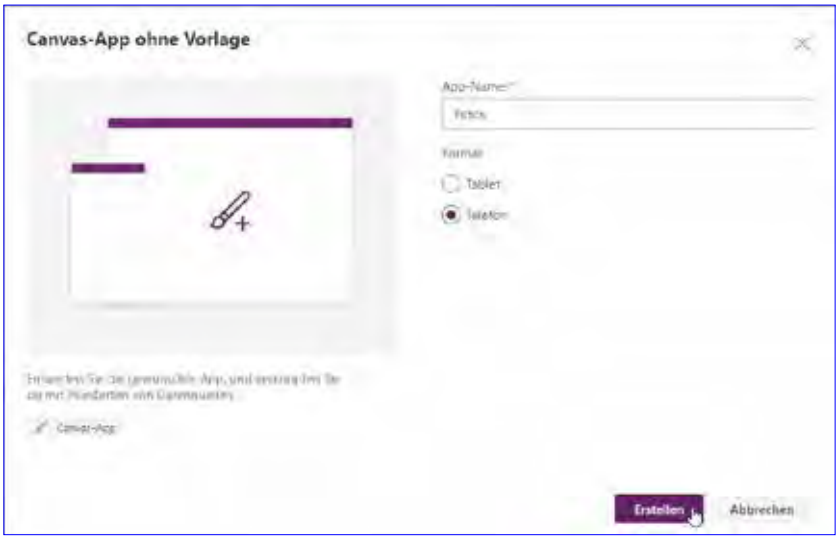


Bild 2: Einstellen der Basiseigenschaften

gegebenenfalls noch die **Veröffentlichen**-Schaltfläche.

Nachdem wir die App dann mangels Smartphone in der PowerApps-App gestartet haben, erscheint immerhin ein Bildschirm, der die Vorschau der Kamera anzeigt. Allerdings finden wir hier keinerlei weitere Bedienelemente wie etwa einen Auslöser oder zum Zoomen. Und auch das Zoomen durch größer- oder kleiner ziehen des Motivs funktioniert nicht. Also benötigen wir wohl noch weitere Elemente, um unsere erste Aufnahme zu schießen.

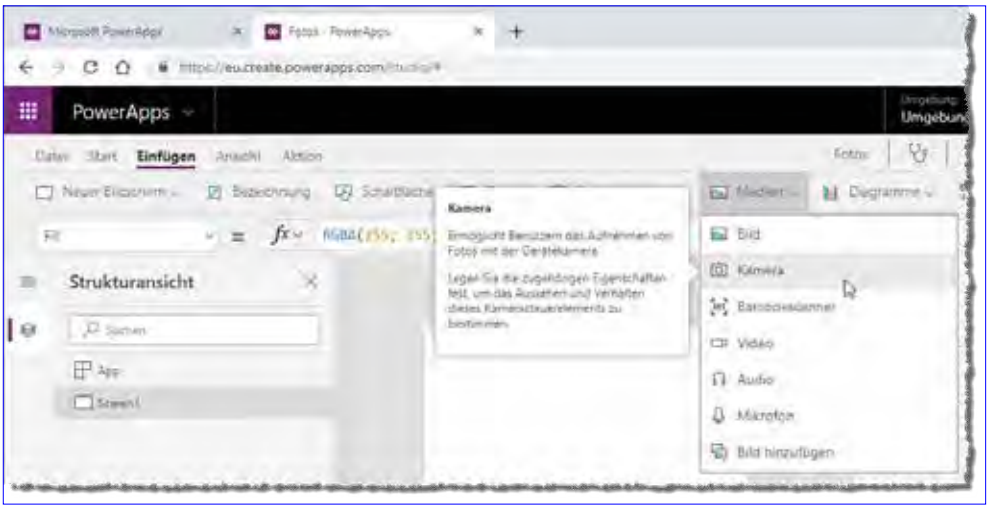


Bild 3: Hinzufügen des Kamera-Elements

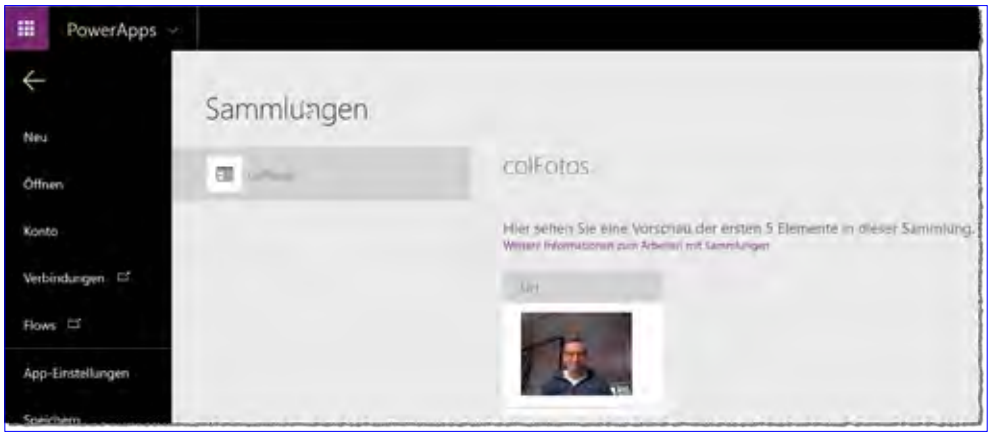


Bild 4: Anzeige eines Bildes aus einer Collection

Testen mit Kamera

Vorab der Hinweis, dass das Entwickeln einer PowerApp mit Kamerafunktion auf einem Rechner ohne angeschlossene Kamera keinen Spaß macht. Sie können die vielen Features, die zur Entwurfszeit zur Verfügung stehen, schlicht nicht nutzen – unter anderem die direkte Anzeige der geschossenen Fotos in der zum Ablegen verwendeten Collections.

Auslösen per Klick auf das Bild

Die einfachste Möglichkeit, ein Foto aufzunehmen, ist ein Klick direkt auf das Kamera-Steuerelement. Dazu fügen Sie der Eigenschaft **OnSelect** des Kamera-Steuerelements im Bereich **Erweitert** der Eigenschaften den folgenden Wert hinzu:

```
ClearCollect(colFotos;Camera1.Photo)
```

Dabei ist **colFotos** eine Sammlung, die direkt nach dem Aufnehmen des ersten Fotos erstellt wird und **Camera1** ist der Name unseres Kamera-Steuerelements. Während des Entwurfs können Sie die Sammlung einsehen, indem Sie oben im Menü den Eintrag **Ansicht** auswählen und dann auf **Sammlungen** klicken. Nach dem Aufnehmen eines ersten Fotos erscheint dieses dann in der Sammlung (siehe Bild 4). Allerdings finden wir dort immer nur das zuletzt aufgenommene Foto vor. Wie können wir mehrere Fotos anzeigen beziehungsweise diese überhaupt erst der Sammlung hinzufügen? Wir haben schlicht die falsche Me-

PowerApps: Bilder in Datenbank speichern

Im Artikel »PowerApps: Kamera steuern« zeigen wir, wie Sie das Kamera-Steuerelement in PowerApps nutzen, um Fotos aufzunehmen und diese entweder direkt in einem Image-Steuerelement anzuzeigen oder in einer Sammlung abzulegen, deren Elemente dann in einem Katalog-Steuerelement angezeigt werden können. Beide Methoden sind nicht als dauerhafter Speicherort zu verstehen, sodass wir uns eine Alternative überlegen müssen. Für Datenbank-Entwickler liegt es nahe, die erfassten Bilddateien in einer SQL Server-Datenbank zu speichern. Wie das gelingt, zeigt der vorliegende Beitrag.

Fotos speichern

Wenn wir mit den Lösungen aus dem Artikel **PowerApps: Kamera steuern** Fotos aufnehmen, landen diese Fotos zwar in einem Bildsteuerelement oder in einer Sammlung, allerdings nur temporär: Wenn wir die PowerApp speichern und veröffentlichen und diese in der PowerApps-App auf unserem Smartphone testen, finden wir die Bilder nach einem Neustart der App nicht mehr in der Liste vor. Also benötigen wir eine alternative Möglichkeit zum Speichern der Bilder.

Es gibt die Möglichkeit, Bilder unter SharePoint zu speichern, in OneDrive oder auch in einer SQL Server-Datenbank. Da wir uns hier mit Datenbanken beschäftigen, wollen wir die Bilder in einer SQL Server-Datenbank speichern.

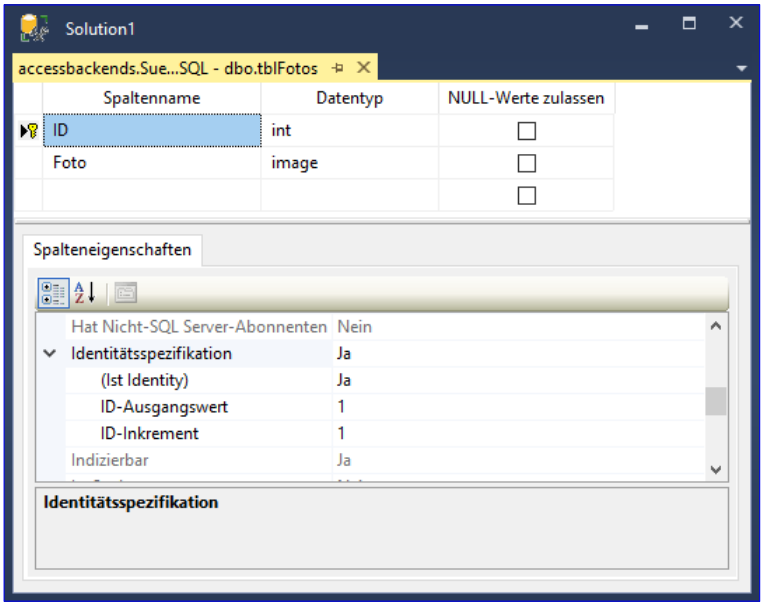


Bild 1: Anlegen der neuen PowerApp

Dazu benötigen wir zunächst eine Beispieldatenbank mit einer Tabelle, die ein Feld des Typs **image** enthält. Außerdem muss die Datenbank natürlich über das Internet verfügbar sein, damit wir auch vom Smartphone aus darauf zugreifen können. Der Einfachheit halber fügen wir unserer Beispieldatenbank **Südsturm**, die auf einem Azure-Server liegt, einfach eine Tabelle namens **tblFotos** hinzu. Diese Tabelle soll neben dem Primärschlüsselfeld **ID** noch ein Feld namens **Foto** erhalten (siehe Bild 1). Die Tabelle erstellen wir beispielsweise im SQL Server Management Studio, mit dem wir uns leicht mit dem Azure-Datenbankserver verbinden können. Wenn Sie noch keine Azure-Datenbank angelegt haben, finden Sie im Artikel **SQL Server-Datenbank ins Web mit SQL Azure** aus Ausgabe 6/2018 alle notwendigen Informationen.

Datenquelle verknüpfen

Nachdem Sie die SQL Server-Datenbank mit der Tabelle **tblFotos** erstellt haben, legen wir eine Verknüpfung zu dieser Tabelle als Datenquelle in der PowerApp an. Dazu wählen Sie im oberen Menü den Eintrag **Ansicht** und im unteren Menü den Eintrag **Datenquellen** aus. Daraufhin erscheint der Bereich **Daten**, der uns den Eintrag **Datenquelle hinzufügen** anbietet (siehe Bild 2).

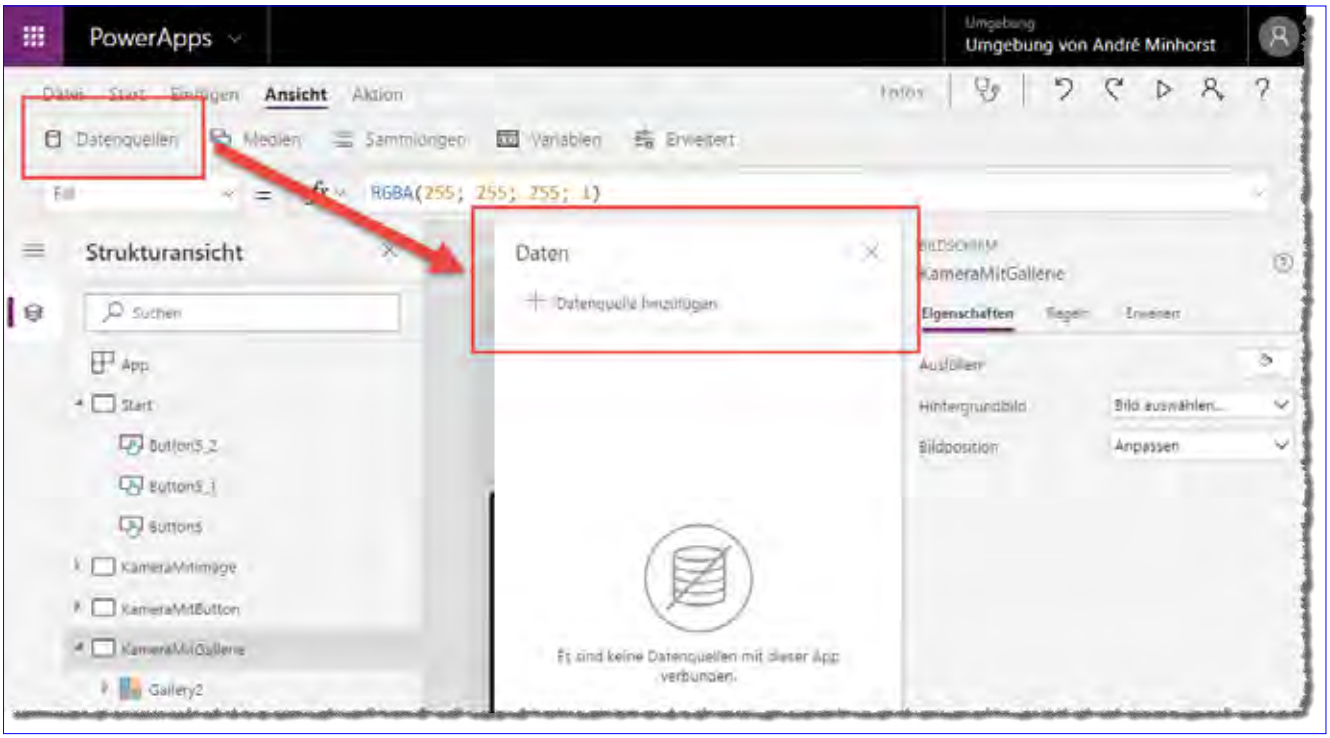


Bild 2: Hinzufügen einer neuen Datenquelle

Die Liste im Bereich **Daten** zeigt nun alle bereits zuvor verwendeten Datenquellen an. Finden Sie die gewünschte Verbindung hier nicht vor, klicken Sie auf **Neue Verbindung**. Es erscheint eine Liste aller möglichen Verbindungen, von denen Sie den Eintrag **SQL Server** auswählen. Nach dem Anklicken dieses Eintrags erscheint ein weiterer Bereich, der Sie zur Eingabe der Verknüpfungsdaten wie **Name des SQL Servers**, **Name der SQL-Datenbank**, **Benutzername** und **Kennwort** auffordert.

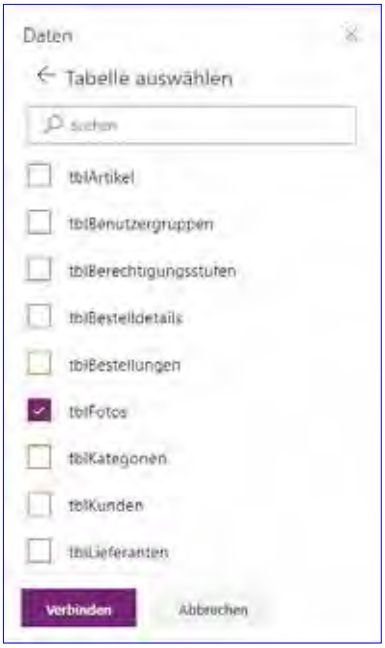


Bild 3: Auswählen der Tabelle zum Speichern der Fotos

Nach der korrekten Eingabe dieser erscheint eine Liste der Tabellen der SQL Server-Datenbank. Hier wählen Sie die Tabelle **tblFotos** aus und betätigen die Schaltfläche **Verbinden** (siehe Bild 3). Die so verknüpfte Tabelle steht dann als Datenquelle bereit.

Bildschirm mit Kamera erstellen

Danach erstellen wir einen neuen Bildschirm mit einem Kamera-Steuerelement namens **cam1** im oberen Bereich. Darunter fügen wir zwei Schaltflächen namens **btnFotoSpeichern** und **btnZurueck** ein. Ganz unten legen wir ein Bildsteuerelement namens **imgFotoDb** an (siehe Bild 4).

Wir belassen es in diesem Beispiel bei der Möglichkeit, Fotos durch Anklicken oder Antippen des Kamera-Steuerelements aufzunehmen und die Fotos dann im Bildsteuerelement **imgFotoDb** anzuzeigen, indem wir dessen Eigenschaft **Image** auf den Wert **cam1.Photo** einstellen.

PowerApps: Bilder aus der Datenbank anzeigen

Im Artikel »PowerApps: Bilder in der Datenbank speichern« zeigen wir, wie Sie Fotos, die Sie mit dem Kamera-Steuerelement einer PowerApp aufgenommen haben, in der Tabelle einer Datenbank speichern. Im vorliegenden Artikel fügen wir der PowerApp eine weitere Bildschirmseite hinzu, auf der wir die Bilder dieser Datenbank anzeigen und verwalten.

Wir setzen auf der Anwendung aus dem Artikel **PowerApps: Bilder in der Datenbank speichern** auf, da wir die dortige Tabelle und die darin über die PowerApp gespeicherten Bilder nutzen wollen.

Der PowerApp aus dieser Anwendung fügen wir eine neue Bildschirmseite hinzu, der wir den Namen Fotos aus der Datenbank geben. Wir verwenden wieder die Vorlage für eine leere Bildschirmseite.

Das Herstellen der Verbindung mit der Datenquelle haben wir bereits im oben genannten Artikel beschrieben, sodass wir nun direkt auf die Tabelle **dbo.tblFotos** als Datenquelle zugreifen können.

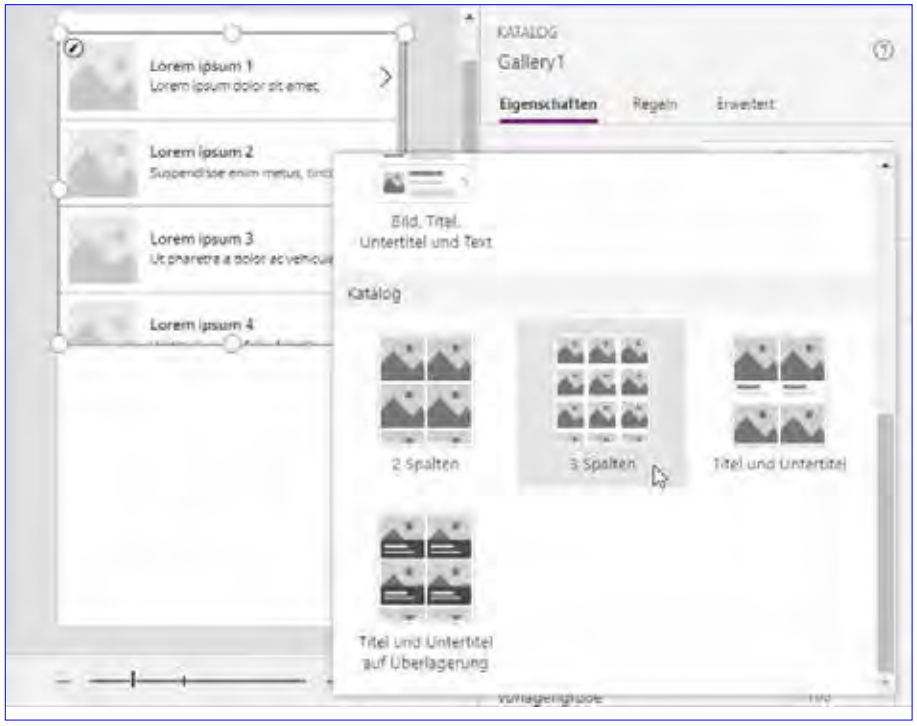


Bild 1: Einstellen des Formats des Katalogs zur Bildanzeige

Anschließend fügen wir der Bildschirmseite einen neuen Katalog hinzu, indem Sie im oberen Menü den Eintrag **Einfügen** und im unteren Menü den Eintrag **KataloglVertikal** auswählen. Dieses Format wollen wir nicht beibehalten, deshalb öffnen wir die Auswahlliste für die Eigenschaft **Layout** im Bereich **Eigenschaften** und wählen dort den Eintrag **3 Spalten** aus (siehe Bild 1).

Im gleichen Bereich wählen wir nun die Tabelle **[dbo].[tblFotos]** für die Eigenschaft **Elemente** aus (siehe Bild 2).

Danach erscheinen die in der Datenbank gespeicherten Bilder bereits in der Katalog-Ansicht (siehe Bild 3).

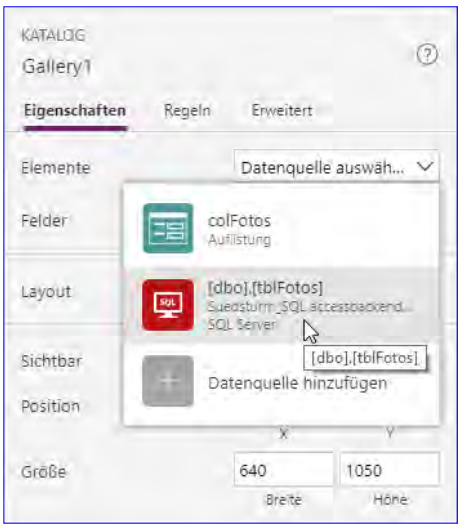


Bild 2: Einstellen des der Datenquelle für den Katalog zur Bildanzeige

Bild im Detail anzeigen

Nun fügen wir eine weitere Bildschirmseite hinzu, welche eines der Bilder nach dem Anklicken in der vollen Ansicht anzeigt. Diese neue Seite nennen wir **FotoAusDerDatenbankDetail**.

Der Seite fügen wir im oberen Bereich eine Schaltfläche namens **btnZurueck** hinzu. Diese statten wir über die Eigenschaft **OnSelect** mit dem Befehl **Back()** aus. Mit dieser Schaltfläche wollen wir nach dem Anzeigen des Bildes in der Detailansicht wieder zur Übersicht zurückkehren.

Dann fügen wir der Bildschirmseite über den Menübefehl **FormularelAnzeigen** ein Formular hinzu, das später ein Bildsteuerelement zum Anzeigen des gewünschten Bildes aufnehmen soll. Nachdem wir das Formular an die Größe des Bildschirms angepasst haben, sieht dieser wie in Bild 4 aus.

Nun legen wir die Datenquelle für das Formular fest. Sie könnten nun auf den Link **Mit Daten verbinden** klicken, erneut die SQL Server-Datenbank auswählen und die Tabelle **tblFotos** festlegen. Dadurch würde aber eine neue Verknüpfung namens **[dbo].[tblFotos_1]** in der App angelegt, was unnötig ist. Also geben Sie schlicht **'[dbo].[tblFotos]'** für die

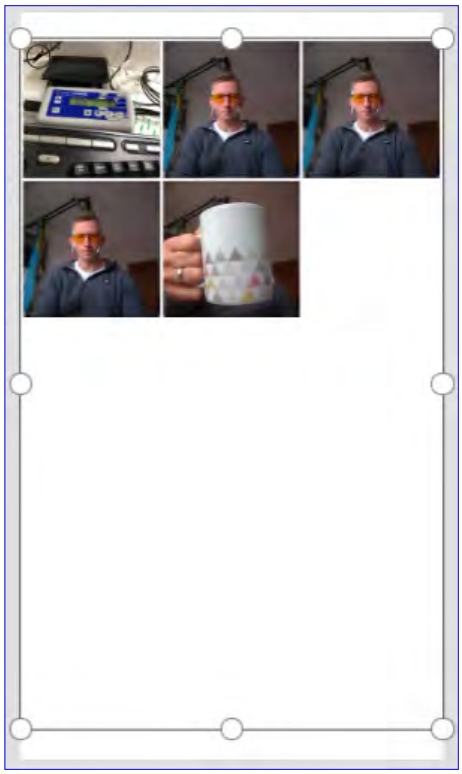


Bild 3: Anzeige von Bildern aus der Datenbank im Katalog-Steuerelement

Bild 4: Detailformular für die Anzeige der Fotos

Bild 5: Einstellen der Eigenschaften für die Datenquelle

Von Access zu Entity Framework: Update 1

In Ausgabe 5/2018 haben wir in zwei Artikeln gezeigt, wie Sie das Datenmodell und die Daten einer Access-Datenbank in ein Entity Data Model und darüber in eine SQL Server-Datenbank migrieren. Im vorliegenden Artikel finden Sie eine Optimierung der dort beschriebenen Prozeduren. Im Detail geht es darum, dass in Access-Tabellen manchmal Namen in Tabellen verwendet werden, die gleichzeitig Plural und Singular der enthaltenen Entität sind – wie zum Beispiel bei tblArtikel. Das führt bei unserer automatisierten Migration früher oder später zu Problemen. Daher erweitern wir unsere Prozeduren um die notwendigen Unterscheidungen.

Im Artikel **Access zu WPF: Detailformulare mit Textfeldern** aus der aktuellen Ausgabe beispielsweise erstellen wir das Code behind-Modul für eine WPF-Detailansicht der Einträge der Tabelle **tblKunden** beziehungsweise der Collection **Kunden** und der Klasse **Kunde**. Hier gelingt alles reibungslos, da wir beim Migrieren der Tabellen der Access-Tabelle in die **DbSet**-Definition und die Klassendefinition mit **Kunde** und **Kunden** entsprechende unterschiedliche Bezeichnungen verwenden konnten. Wenn wir die dort erarbeiteten Prozeduren nutzen wollen, um eine Detailansicht auf Basis eines Formulars zu erstellen, dass die Tabelle **tblArtikel** als Datensatzquelle verwendet, gelingt das nicht problemlos, da beim Migrieren des Datenmodells in das Data Entity Model aus **tblArtikel** sowohl das **DbSet**-Element **Artikel** als auch die Klasse **Artikel** generiert wurden. Das führt noch nicht zu Fehlern, aber wenn wir dann mit der Prozedur **FormularNachWPF_CodeBehind** die Code behind-Klasse erzeugen und dort einfügen, erhalten wir eine Reihe von Fehlermeldungen (siehe Bild 1).

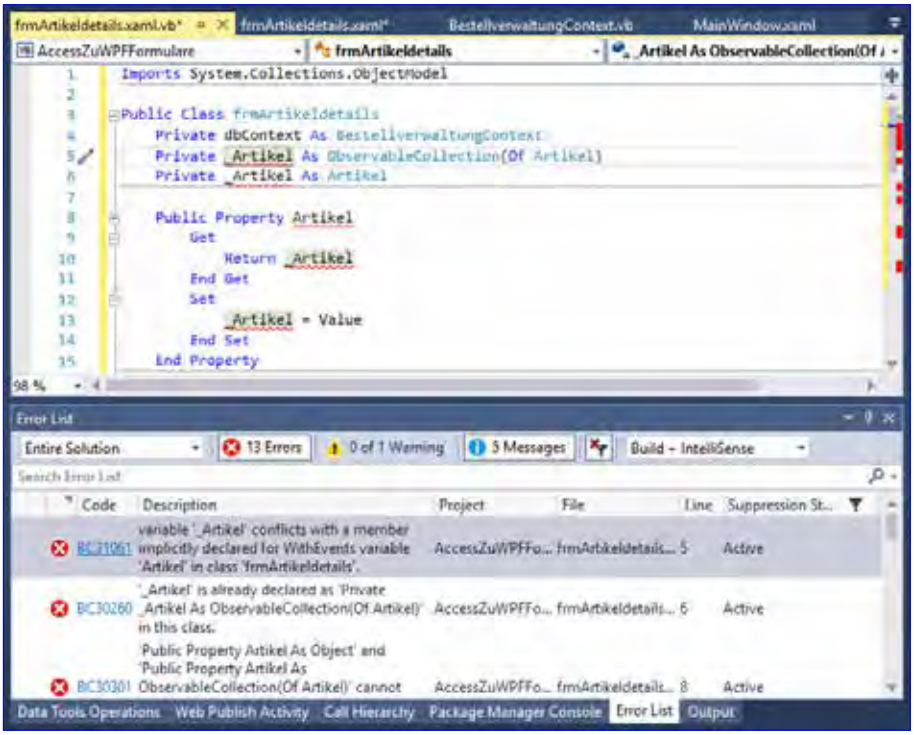


Bild 1: Fehler bei Verwendung gleicher Variablennamen für Objekte und Collections

Wie können wir das Problem lösen? Eine Möglichkeit wäre, die Bezeichnungen im Code etwa der Code behind-Klasse so anzupassen, dass die verschiedenen mehrfach vorkommenden Variablen und Eigenschaftsbezeichnungen eindeutige Namen erhalten. Noch schöner wäre es allerdings, wenn wir direkt, wie auch beim Beispiel der Tabelle **tblKunden**, eindeutige Namen für die Klasse und die **DbSet**-Elemente hätten – in diesem Fall auch der besseren Lesbarkeit halber als Singular und Plural.

Erstellung des Entity Data Models anpassen

Und hier wollen wir ansetzen, indem wir die Prozedur **EDMErstellen** des Moduls **mdlEDM** aus dem Artikel **Von Access zu Entity Framework: Datenmodell** entsprechend anpassen. Das sieht dann ausschnittsweise wie folgt aus:

```
Public Sub EDMErstellen()
    ...
    For Each tdf In db.TableDefs
        If Not Left(tdf.Name, 4) = "MSys" Then
            strPK = ""
            If Not GetPrimaryKey(tdf.Name, strPK) Then
               ...
            Else
                strEntity = Replace(strPK, "ID", "")
                strEntities = Replace(tdf.Name, "tbl", "")
                If strEntity = strEntities Then
                    Do While strEntity = strEntities
                        MsgBox "Plural und Singular der Bezeichnung der Entitäten (hier """ & strEntity _
                            & """) im Tabellennamen scheinen gleich zu sein." & vbCrLf & vbCrLf & "Geben Sie " _
                            & "nachfolgend eine Bezeichnung für den Singular der Entität ein und eine für den Plural."
                        strEntity = InputBox("Bezeichnung für den Singular/den Klassennamen der Entität """ _
                            & strEntity & """:", "Singular bestimmen", strEntity)
                        strEntities = InputBox("Bezeichnung für den Plural/den Namen der Auflistung der Entität """ _
                            & strEntities & """:", "Plural bestimmen", strEntities)
                    Loop
                End If
                ...
            End If
        End If
    Next tdf
    ...
End Sub
```

Nachdem wir die Bezeichnungen für die einzelne Entität und die Mehrzahl ermittelt und in **strEntity** und **strEntities** gespeichert haben, vergleichen wir diese in einer **If... Then**-Bedingung. Sind beide gleich, steigen wir in eine **Do While**-Schleifen ein. Diese zeigt zunächst eine Meldung wie in Bild 2 an.

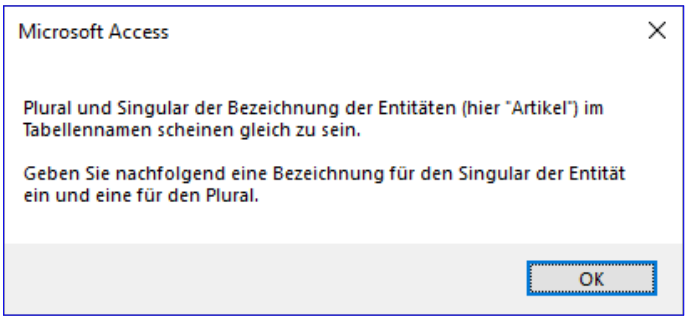


Bild 2: Meldung bei Gleichheit von Singular und Plural

Nach dieser Erklärung folgen zwei **InputBox**-Anweisungen, mit denen die Prozedur die Singular- und die

Plural-Version für die zu erstellenden Klassen und **DbSet**-Elemente abfragt. Sind die beiden Werte von **strEntity** und **strEntities** nach dem Durchlauf der **Do While**-Schleife nicht mehr gleich, wird diese verlassen und die Prozedur erstellt die Klasse und die **DbSet**-Definition mit den neuen Namen.

Wenn wir die Prozedur nun aufrufen und die erstellten und in die Zwischenablage kopierten Code-Strukturen in das Projekt einfügen, finden wir allerdings noch einen kleinen Syntaxfehler vor: Das Fremdschlüsselfeld **ArtikelID** der Tabelle **tblArtikel** wird noch nicht entsprechend der neuen Benennung der **Artikel**-Entitäten umbenannt. Das Feld sollte nun **ProduktID** statt **ArtikelID** heißen und die Eigenschaft, die das **Artikel**-Objekt aufnimmt, sollte das **Produkt**-Objekt aufnehmen (siehe Bild 3).

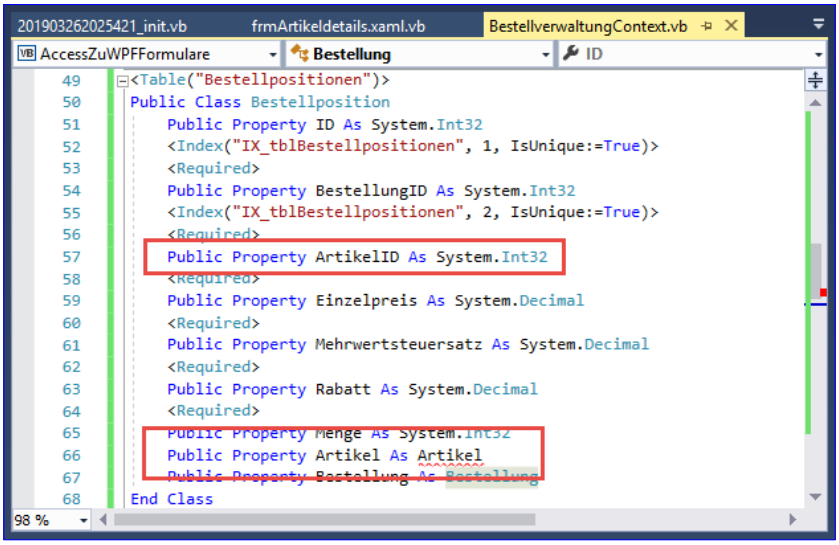


Bild 3: Artikel muss noch durch Produkt ersetzt werden.

Da wir in der Prozedur alle Tabellen in willkürlicher Reihenfolge durchlaufen, kann es sein, dass wir die Tabelle **tblArtikel** erst nach der Tabelle **tblBestelldetails** durchlaufen. Wir erfahren aber erst beim Durchlaufen der Tabelle **tblArtikel**, dass hier eine Umbenennung notwendig ist, die sich auch schon auf Felder der Tabelle **tblBestelldetails** auswirkt. Wir müssen also eine Schleife einbauen, die schon vorher prüft, ob Ersetzungen bei Tabellennamen notwendig sind, weil dieser gleichzeitig Plural und Singular ist und dann später beim zweiten Durchlauf und beim Zusammenstellen des Codes für die Entitäten und **DbSet**-Elemente auf die hier gewonnenen Informationen zurückgreifen.

Dazu sind einige umfangreichere Umstellungen nötig. Diese beruhen im Wesentlichen darauf, dass wir die **TableDef**-Objekte nicht mehr einfach in einer Schleife durchlaufen und dabei die Definitionen aller Entitäten zusammenstellen können – aus dem obigen Grund müssen wir erst einmal alle Tabellen dahingehend analysieren, ob sie Entitätsnamen hervorbringen würden, die gleichzeitig Plural und Singular sind. Also durchlaufen wir zunächst in einer Schleife alle **TableDef**-Objekte und tragen die relevanten Informationen jeweils in eine Klasse ein, die entsprechende Eigenschaften aufnimmt. Die neue Klasse legen Sie im VBA-Editor mit dem Menübefehl **Einfügen|Klassenmodul** an und speichern diese unter dem Namen **clsMapping**. Die folgenden privaten Variablen speichern die Werte:

```
Private m_Tabelle As String
Private m_PK As String
Private m_Entity As String
Private m_Entities As String
Private m_ID As String
Private m_Entity_Original As String
Private m_Entities_Original As String
```

Für jeden dieser Werte gibt es je eine **Property Get**- und eine **Property Let**-Methode, die wie folgt aussehen – hier stellvertretend für die Eigenschaft **Tabelle**:

```
Public Property Get Tabelle() As String
    Tabelle = m_Tabelle
End Property

Public Property Let Tabelle(str As String)
    m_Tabelle = str
End Property
```

Wir erstellen für jedes **TableDef**-Objekt eine dieser Klassen und speichern diese in einer **Collection** namens **colMappings**, die wir wie folgt deklarieren und mit dem Schlüsselwort **New** erstellen:

```
Public Sub EDMErstellen()
    Dim db As DAO.Database, tdf As DAO.TableDef, fld As DAO.Field
    Dim strDbSets As String, strPK As String, strDatatype As String
    Dim bolArray As Boolean, bolDatatypeExists As Boolean
    Dim strArray As String, strFieldname As String, strUntertyp As String
    Dim strIndex As String, strCode As String, colMappings As Collection
    Dim objMapping As clsMapping
    Dim objMappingFK As clsMapping
    Set db = CurrentDb
    Set colMappings = New Collection
```

Dann beginnen wir, die **TableDefs** zu durchlaufen. Dabei prüfen wir wieder, ob der Tabellenname nicht mit **MSys** oder **~** beginnt. Dann prüfen wir, ob die Tabelle einen Primärschlüssel aufweist. Falls nicht, landet ein entsprechender Kommentar in unserem in **strCode** zusammengestellten Code für das VB-Projekt:

```
    For Each tdf In db.TableDefs
        If Not Left(tdf.Name, 4) = "MSys" And Not Left(tdf.Name, 1) = "~" Then
            If Not GetPrimaryKey(tdf.Name, strPK) Then
                strCode = strCode & "    '****" & vbCrLf
                strCode = strCode & "    'Mehrere PKs in " & tdf.Name & ". Die Klasse wird nicht erstellt." & vbCrLf
```

Falls die Tabelle ein Primärschlüsselfeld besitzt, tragen wir einige Werte in die Eigenschaften des Objekts **objMapping** ein, das wir dazu neu erstellen. **Entity** wird mit dem Primärschlüsselwert ohne **ID** gefüllt, **Entity_Original** mit dem gleichen Wert. Gleiches gilt für **Entities** und **Entities_Original**, die mit dem Tabellennamen ohne **tbl** gefüllt werden. Sind **.Entity** und **.Entities** gleich, durchlaufen wir die schon oben beschriebene Schleife, in welcher der Benutzer solange neue Werte für **.Entity** und **.Entities** eingibt, bis sich beide unterschieden (etwa **Produkt** und **Produkte** statt **Artikel**). Ist das erledigt, schreiben wir noch **ID** in die Eigenschaft **.ID**, den Primärschüsselnamen in **.PK** und den Tabellennamen in **.Tabelle**:

```
        Else
            Set objMapping = New clsMapping
            With objMapping
                .Entity = Replace(strPK, "ID", "")
                .Entity_Original = .Entity
                .Entities = Replace(tdf.Name, "tbl", "")
                .Entities_Original = .Entities
                If .Entity = .Entities Then
                    Do While .Entity = .Entities
                        MsgBox "Plural und Singular der Bezeichnung der Entitäten (hier """ & .Entity _
                            & """) im Tabellennamen scheinen gleich zu sein." & vbCrLf & vbCrLf & "Geben Sie " _
                            & "nachfolgend eine Bezeichnung für den Singular der Entität ein und eine für den Plural."
                        .Entity = InputBox("Bezeichnung für den Singular/den Klassennamen der Entität """ & .Entity _
                            & """:", "Singular bestimmen", .Entity)
                        .Entities = InputBox("Bezeichnung für den Plural/den Namen der Auflistung der Entität """ _
                            & .Entities & """:", "Plural bestimmen", .Entities)
                    Loop
                End If
                .ID = "ID"
                .PK = strPK
                .Tabelle = tdf.Name
            End With
        End If
```

Schließlich fügen wir das Objekt **objMapping** zu **colMappings** hinzu, damit wir es später nutzen können:

```
            colMappings.Add objMapping
        End If
    Next tdf
```

Und das geschieht recht schnell, denn gleich nach dem Einlesen der relevanten Informationen durchlaufen wir alle Elemente der **colMappings**-Collection in einer **For Each**-Schleife, wobei wir das jeweilige Objekt mit **objMapping** referenzieren:

```
    For Each objMapping In colMappings
```

Innerhalb dieser Schleife ersetzen wir alle Referenzen auf **strEntity**, **strEntities** und so weiter durch Referenzen auf die entsprechenden Elemente des Objekts **objMapping**. Daraus holen wir das **TableDef**-Objekt mit dem Namen aus **.Tabelle**, um dessen Felder über die **Fields**-Auflistung zu durchlaufen. Vorher stellen wir noch die Zeile für das **DbSet**-Element sowie die Kopfzeile der Entität zusammen:

```
        With objMapping
```

Access zu WPF: Detailformulare mit Textfeldern

Wir haben uns bereits in einigen Artikel angesehen, wie Sie Übersichtsformulare, Detailformulare und so weiter unter WPF anlegen. Was aber, wenn Sie keine Lust haben, die Formulare unter WPF alle neu zu programmieren, obwohl Sie das schon unter Access erledigt haben? In diesem Artikel schauen wir uns an, wie die programmgesteuerten Möglichkeiten aussehen, um Formulare automatisch als WPF-Fenster oder -Seiten abzubilden. Es wird ein wenig Handarbeit übrig bleiben, aber einen großen Teil der Schritte können Sie sich damit deutlich erleichtern.

Voraussetzungen

Wir gehen an dieser Stelle davon aus, dass Sie bereits ein Entity Data Model auf Basis des Access-Datenmodells erstellt und eine entsprechende SQL Server-Datenbank auf Basis des Entity Data Modells erstellt haben. Wie das gelingt, zeigen die Artikel **Von Access zu Entity Framework: Datenmodell** und **Von Access zu Entity Framework: Daten** aus Ausgabe 5/2018.

Nun wollen wir versuchen, die Migration einfacher Formulare von Access aus in funktionsfähige WPF-Fenster zu erreichen.

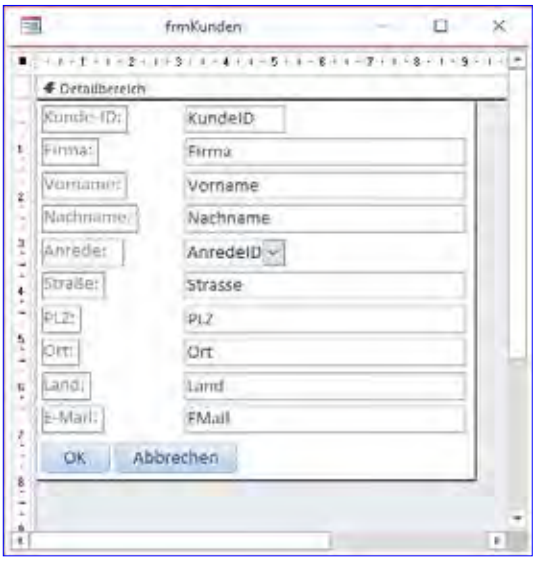


Bild 1: Das Formular **frmKunden** in der Formularansicht

Beispiel: Kundenformular

Unser Beispiel finden Sie in der Beispieldatenbank **Bestellverwaltung.accdb** im Download zu diesem Artikel. Wir wollen das Formular **frmKundendetails** als WPF-Fenster nachbauen. Das Formular sehen Sie in Bild 1 in der Entwurfsansicht. Es verwendet die Tabelle **tblKunden** als Datensatzquelle. Die Textfelder sind jeweils an entsprechende Felder der Datensatzquelle des Formulars gebunden. Das Kombinationsfeld **cboAnredeID** verwendet die Abfrage **SELECT [tblAnreden].[AnredeID], [tblAnreden].[Anrede] FROM tblAnreden;** als Datensatzherkunft und ist an das Feld **AnredeID** gebunden. Für das erste Feld der Datensatzherkunft haben wir die Spaltenbreite auf **0cm** eingestellt, damit dieses nicht angezeigt wird. Es erscheint also nur das zweite Feld der Datensatzherkunft.

Projekt erstellen

Wir erstellen ein neues Projekt namens **AccessZuWPFFormulare** und führen die im Artikel **Von Access zu EF: Step by step** aus Ausgabe 5/2018 beschriebenen Schritte aus, um die Tabellenentwürfe und Daten der Tabellen unserer Beispieldatenbank als Entity Data Model hinzuzufügen. Das gelingt innerhalb von ungefähr fünf Minuten, wenn Sie die Beispieldatenbank dazu verwenden. Für andere Datenbanken haben wir dies noch nicht getestet. Wenn Sie versuchen, andere Datenbanken zu migrieren und auf Fehler stoßen, teilen Sie uns das gern mit. Wir benötigen dann allerdings eine Kopie des Datenmodells im Access-Format, damit wir damit experimentieren und unsere Tools verbessern können (per E-Mail an **andre@minhorst.com**).

Zum Testen der migrierten Datenbank verwenden Sie eine neue Schaltfläche auf der Seite **MainWindow.xaml**, welche die folgende Methode aufruft:

```
Private Sub btnTestDatabase_Click(sender As Object, e As RoutedEventArgs)
    Dim dbContext As BestellverwaltungContext
    Dim Kunden As ObservableCollection(Of Kunde)
    dbContext = New BestellverwaltungContext
    Kunden = New ObservableCollection(Of Kunde)(dbContext.Kunden)
    MessageBox.Show("Anzahl Kunden: " + Kunden.Count().ToString())
End Sub
```

Danach sollte beim Starten des Projekts und Anklicken der Schaltfläche ein Meldungsfenster mit der Anzahl der Datensätze der Tabelle **Kunden** erscheinen.

Vom Formular zu WPF

Nun schauen wir uns an, was für eine Aufgabe überhaupt vor uns liegt. Wir wollen das Formular **frmKundendetails** aus der Access-Datenbank als Fenster der WPF-Anwendung abbilden. Was benötigen wir dazu? Zunächst einmal ist soll das Fenster die Größe erhalten, die auch unser Access-Formular aufweist. Dann wollen wir die Steuerelemente auslesen und im WPF-Fenster anlegen. Diese sollen, genau wie das Fenster, mit den Eigenschaften zum Binden des Fensters und der Steuerelemente an die Daten versehen werden. Schließlich fehlen noch die Ereignisprozeduren, die durch die Steuerelemente und das Formular selbst ausgelöst werden.

Wir wollen zunächst von Access aus starten und hier Code schreiben, mit dem wir die notwendigen Informationen aus dem Formularentwurf auslesen und daraus automatisiert die XAML-Definition für das WPF-Fenster sowie den Inhalt der Code behind-Daten für das Fenster erstellen. Diesen kopieren wir dann zunächst von Hand in ein neues Fenster-Element in unserem Projekt.

Fenstergröße

Wir beginnen einmal mit einer vermeintlich einfachen Aufgabe: dem Ermitteln der Fenstergröße und dem Erstellen des Code-Elements, welches diese beiden Eigenschaften enthält. Wenn wir die Eigenschaften im Eigenschaftenblatt von Access ansehen, erhalten wir Angaben in der Einheit Zentimeter, also beispielsweise **9,198cm** für die Formularbreite und **9,198cm** für die Höhe des Detailbereichs. Geben wir die Eigenschaften **Width** des Form-Elements und **Me.Height** des Detailbereichs (**Me.Section(0).Height**) aus, erhalten wir Werte wie **4535** und **5215**. Wenn wir die Eigenschaften in der XAML-Definition betrachten, finden wir Werte wie **450** für die Eigenschaft **Height** und **800** für die Eigenschaft **Width**. Wir müssen die Zahlen also noch umrechnen. Die Größe und Breite unter Visual Studio wird in Pixel angegeben. Unter Access werden diese Werte in Twips angegeben. Zur Umrechnung von Twips in Pixel gibt es ausreichend Funktionen, von denen wir zwei im Modul **mdlUmrechnungen** der Beispieldatenbank eingefügt haben. Sie heißen **TwipsToPixelHeight** und **TwipsToPixelWidth**. Wir benötigen zwei Funktionen, weil die Faktoren für die Umrechnungen unterschiedlich sein können.

Basisprozedur

Damit legen wir nun die Basis für das Erstellen des WPF-Codes. Die benötigte VBA-Prozedur sieht wie folgt aus, wobei wir zunächst einige Variablen deklarieren:

```
Public Sub FormularNachWPF()
    Dim frm As Form
    Dim strForm As String
    Dim strXAML As String
    Dim lngHeight As Long
    Dim lngWidth As Long
    Dim strTitle As String
```

Dann öffnen wir das Formular in der Entwurfsansicht und weisen dieses der Variablen **frm** zu:

```
    strForm = "frmKundendetails"
    DoCmd.OpenForm strForm, acDesign
    Set frm = Forms(strForm)
```

Dann ermitteln wir mit den Umrechnungsfunktionen die Höhe und Breite:

```
    lngHeight = TwipsToPixelsHeight(frm.Section(0).Height)
    lngWidth = TwipsToPixelswidth(frm.Width)
```

Außerdem prüfen wir, ob ein Titel für das Formular angegeben wurde und schreiben diesen oder alternativ den Formularnamen in die Variable **strTitle**:

```
    If Not Len(frm.Caption) = 0 Then
        strTitle = frm.Caption
    Else
        strTitle = frm.Name
    End If
```

Schließlich bauen wir den grundlegenden XAML-Code für das Fenster zusammen, wobei wir den standardmäßig verwendeten Code nutzen und unsere ermittelten Werte für Fenstername, Titel, Höhe und Breite an den entsprechenden Stellen einfügen:

```
    strXAML = strXAML & "<Window x:Class=""" & frm.Name & """" & vbCrLf
    strXAML = strXAML & "        xmlns=""http://schemas.microsoft.com/winfx/2006/xaml/presentation""" & vbCrLf
    strXAML = strXAML & "        xmlns:x=""http://schemas.microsoft.com/winfx/2006/xaml""" & vbCrLf
    strXAML = strXAML & "        xmlns:d=""http://schemas.microsoft.com/expression/blend/2008""" & vbCrLf
    strXAML = strXAML & "        xmlns:mc=""http://schemas.openxmlformats.org/markup-compatibility/2006""" & vbCrLf
    strXAML = strXAML & "        xmlns:local=""clr-namespace:AccessZuWPFFormulare""" & vbCrLf
    strXAML = strXAML & "        mc:Ignorable=""d""" & vbCrLf
    strXAML = strXAML & "        Title=""" & strTitle & """ Height=""" & lngHeight & """ Width=""" & lngWidth & """>" & vbCrLf
    strXAML = strXAML & "    <Grid>" & vbCrLf
```

Dazwischen platzieren wir den Aufruf der Prozedur **SteuerelementeHinzufuegen**, die wir im Anschluss erstellen – für den ersten Testlauf bleibt diese auskommentiert:

```
    'SteuerelementeHinzufuegen frm, strXAML
    strXAML = strXAML & "    </Grid>" & vbCrLf
    strXAML = strXAML & "</Window>" & vbCrLf
```

Schließlich nutzen wir die Prozedur **InZwischenablage**, um den Inhalt der String-Variablen **strXAML** in die Zwischenablage zu kopieren (siehe **mdlZwischenablage**) und schließen das Formular wieder:

```
    InZwischenablage strXAML
    DoCmd.Close acForm, strForm
End Sub
```

Wenn Sie diese Prozedur ausführen und den Inhalt der Zwischenablage als XAML-Code eines neuen Fensters einsetzen, erhalten Sie die Ansicht aus Bild 2.

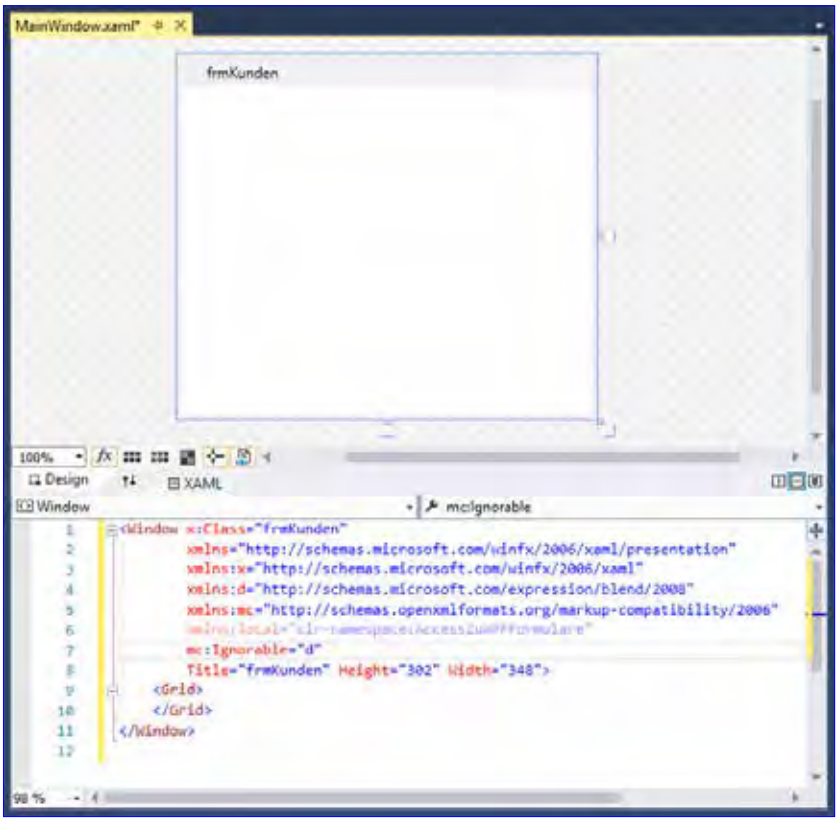


Bild 2: Höhe, Breite, Titel und Name wurden vom Access-Formular übernommen

Ob dieses Fenster unseren Ansprüchen genügt, müssen wir allerdings noch herausfinden – dazu benötigen wir im Fenster **MainWindows.xaml** noch eine Schaltfläche, mit der wir das neu erstellte Fenster öffnen können. Diese soll den folgenden Code ausführen:

```
Private Sub btnKundendetails_Click(sender As Object, e As RoutedEventArgs)
    Dim Kundendetails As frmKundendetails
    Kundendetails = New frmKundendetails()
    Kundendetails.Show()
End Sub
```

Steuerelemente migrieren

Als nächstes wollen wir uns um die Steuerelemente kümmern. Den folgenden Aufruf haben wir ja schon im Code platziert – Sie müssen nur noch die Markierung als Kommentar entfernen:

```
   SteuerelementeHinzufuegen frm, strXAML
```

Die Prozedur selbst erwartet die beiden übergebenen Parameter, nämlich einen Verweis auf das Formular (**frm**) und die Variable zum Zusammenstellen des XAML-Codes (**strXAML**). Damit starten wir in die erste Rohfassung der Prozedur:

```
Public Function SteuerelementeHinzufuegen(frm As Form, strXAML As String)
    Dim ctl As Control
    Dim lngWidth As Long
    Dim lngHeight As Long
    Dim lngTop As Long
    Dim lngLeft As Long
    Dim strCaption As String
```

In dieser durchlaufen wir alle Steuerelemente des Formulars über die **Controls**-Auflistung und referenzieren das aktuelle Steuerelement jeweils mit der Variablen **ctl**. Hier tragen wir zunächst die Größe und die Position in die entsprechenden Variablen ein:

```
    For Each ctl In frm.Controls
        lngWidth = TwipsToPixelsWidth(ctl.Width)
        lngHeight = TwipsToPixelsHeight(ctl.Height)
        lngTop = TwipsToPixelsHeight(ctl.Top)
        lngLeft = TwipsToPixelsWidth(ctl.Left)
```

Dann prüfen wir in einer **Select Case**-Schleife, welchen Typ das aktuell mit **ctl** referenzierte Steuerelement hat. In diesem Fall haben wir zunächst die Typen **acTextBox** und **acLabel** untersucht und fügen für jedes Bezeichnungsfeld und jedes Textfeld eine Zeile zu **strXAML** hinzu, welche das Aussehen des jeweiligen Steuerelements beschreibt:

```
        Select Case ctl.ControlType
            Case acTextBox
                strXAML = strXAML & "<TextBox x:Name=""" & ctl.Name & """ FontFamily=""Calibri"" FontSize=""" _
                    & ctl.FontSize & "pt"" HorizontalAlignment=""Left"" Height=""" & lngHeight & """ Margin=""" _
                    & lngLeft & "," & lngTop & ",0,0"" TextWrapping=""Wrap"" Text=""TextBox"" " _
                    & "VerticalAlignment=""Top"" Width=""" & lngWidth & """/>" & vbCrLf
            Case acLabel
               strCaption = ctl.Caption
                strXAML = strXAML & "<Label x:Name=""" & ctl.Name & """ Content=""" & strCaption _
                    & """ FontFamily=""Calibri"" FontSize=""11pt"" Padding=""0"" " _
                    & "VerticalContentAlignment=""Center"" HorizontalAlignment=""Left"" Height=""" & lngHeight _
                    & """ Margin=""" & lngLeft & "," & lngTop & ",0,0"" VerticalAlignment=""Top"" Width=""" & lngWidth _
                    & """/>" & vbCrLf
            Case acCommandButton
            Case acComboBox
            Case Else
```

```
                Debug.Print "Steuerelementtyp nicht behandelt: " & ctl.Name, ctl.ControlType
        End Select
    Next ctl
End Function
```

Die übrigen Steuerelemente schauen wir uns später an. Wenn wir die Prozedur **FormularNachWPF** nun erneut ausführen, erhalten wir Code, der das Fenster aus Bild 3 erstellt.

Bild 3: Textfelder und Bezeichnungsfelder

Zwischenergebnis prüfen

Nachdem wir nun das Fenster und die enthaltenen Steuerelemente definiert haben, können wir die Anwendung einmal starten und das Fenster **frmKundendetails.xaml** einmal über das geöffnete Access-Formular **frmKundendetails** legen. Das Ergebnis sieht wie in Bild 4 aus. Hier stellen wir fest, dass etwas mit der Berechnung von Höhe und Breite des Fensters nicht funktioniert hat. Das WPF-Fenster ist insgesamt kleiner als der Innenraum des Formulars. Berechnet haben wir eine Breite von 348 Pixeln und Höhe von 302 Pixeln. Erstellen wir einen Screenshot und schauen uns die Dimensionen im Screenshotprogramm an, finden wir Maße von 334 x 295 Pixeln vor. Schauen wir uns das Verhältnis für das Fenster **MainWindow.xaml** an, finden wir den gleichen absoluten Fehler – es fehlen 14 Pixel in der Breite und 7 in der Höhe. Legen wir das Fenster nun erneut über das Formular, erkennen wir, dass es den Innenraum komplett abdeckt. Allerdings müssen wir noch einige Pixel für die Titelleiste des WPF-Fensters hinzurechnen – dann passen auch die Schaltflächen noch unter die gebundenen Steuerelemente. Wir messen das grob im Screenshotprogramm und erhalten 23 Pixel, die wir noch zur Höhe hinzuaddieren.

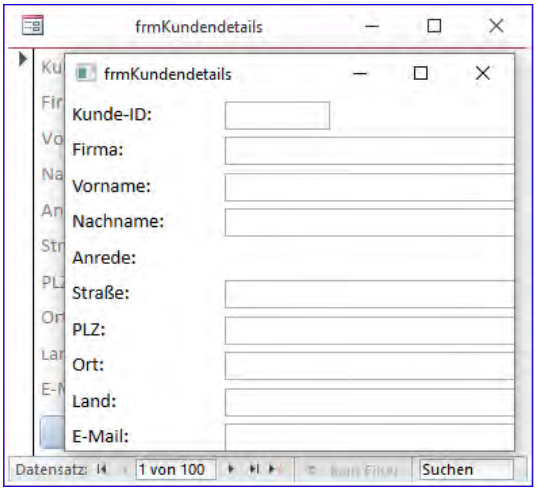


Bild 4: Die Größe von Original und Kopie stimmt nicht überein.

Standardeigenschaften fensterweit definieren

Wenn wir uns die beiden folgenden Elemente ansehen, erkennen wir einige Eigenschaften, die wir für die gleichen Steuerelementtypen immer wieder verwenden werden – zum Beispiel die Schriftart oder die Ausrichtung:

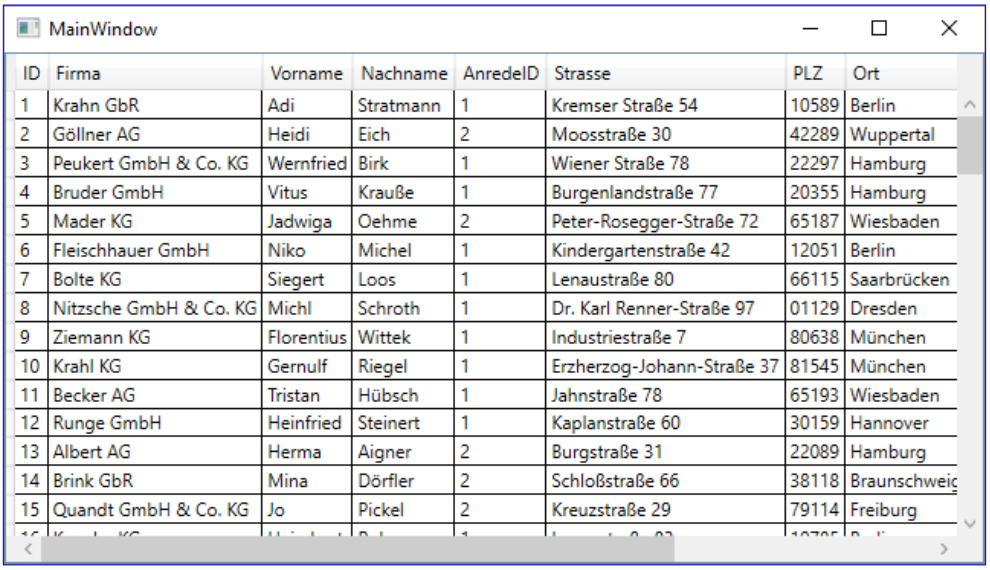
```
<TextBox x:Name="txtKundeID" FontFamily="Calibri" FontSize="11pt" HorizontalAlignment="Left" Height="21" Mar-
gin="118,5,0,0" TextWrapping="Wrap" Text="TextBox" VerticalAlignment="Top" Width="78"/>
<Label x:Name="Bezeichnungsfeld0" Content="Kunde-ID:" FontFamily="Calibri" FontSize="11pt" Padding="0" VerticalContentAli
gnment="Center" HorizontalAlignment="Left" Height="21" Margin="5,5,0,0" VerticalAlignment="Top" Width="67"/>
```

EF: Daten abfragen mit VB und LINQ

Unter Access waren Sie es gewöhnt, auf einfache Weise Abfragen mit der Abfrage-Entwurfsansicht zu erstellen. Ein paar Tabellen hinzufügen, die Felder auswählen, Kriterien, Sortierungen und Gruppierungen hinzufügen – fertig war die Abfrage. Gegebenenfalls haben Sie SQL-Kenntnisse und konnten SQL-Anweisungen für den Einsatz in VBA-Anweisungen von Hand schreiben. Unter VB und Entity Framework sieht das anders aus, weil wir ja nicht mehr auf Tabellen zugreifen, sondern auf Objekte. Und für die gibt es eine andere Abfragesprache, die sich direkt in den VB-Code integrieren lässt. Dieser Artikel stellt die Abfragetechnik LINQ für Visual Basic vor.

Voraussetzungen

Um die Beispiele dieses Artikels nachstellen zu können, benötigen das Entity Data Model aus dem Beispielprojekt aus dem Download zu diesem Artikel. Für die Beispielabfragen fügen wir dem Fenster **MainWindow.xaml** unserer Beispielanwendung ein **DataGrid**-Steuerelement hinzu. Der Code zum Füllen des **DataGrid**-Elements mit allen Eigenschaften aller **Kunden**-Elemente wie in Bild 1 sieht so aus:



ID	Firma	Vorname	Nachname	AnredeID	Strasse	PLZ	Ort
1	Krahn GbR	Adi	Stratmann	1	Kremser Straße 54	10589	Berlin
2	Göllner AG	Heidi	Eich	2	Moosstraße 30	42289	Wuppertal
3	Peukert GmbH & Co. KG	Wernfried	Birk	1	Wiener Straße 78	22297	Hamburg
4	Bruder GmbH	Vitus	Krauße	1	Burgenlandstraße 77	20355	Hamburg
5	Mader KG	Jadwiga	Oehme	2	Peter-Rosegger-Straße 72	65187	Wiesbaden
6	Fleischhauer GmbH	Niko	Michel	1	Kindergartenstraße 42	12051	Berlin
7	Bolte KG	Siegert	Loos	1	Lenaustraße 80	66115	Saarbrücken
8	Nitzsche GmbH & Co. KG	Michl	Schroth	1	Dr. Karl Renner-Straße 97	01129	Dresden
9	Ziemann KG	Florentius	Wittek	1	Industriestraße 7	80638	München
10	Krahl KG	Gernulf	Riegel	1	Erzherzog-Johann-Straße 37	81545	München
11	Becker AG	Tristan	Hübsch	1	Jahnstraße 78	65193	Wiesbaden
12	Runge GmbH	Heinfried	Steinert	1	Kaplanstraße 60	30159	Hannover
13	Albert AG	Herma	Aigner	2	Burgstraße 31	22089	Hamburg
14	Brink GbR	Mina	Dörfler	2	Schloßstraße 66	38118	Braunschweig
15	Quandt GmbH & Co. KG	Jo	Pickel	2	Kreuzstraße 29	79114	Freiburg

Bild 1: DataGrid mit allen Kundendaten

```
Imports System.Collections.ObjectModel

Class MainWindow
    Private dbContext As BestellverwaltungContext
    Private _Kunden As ObservableCollection(Of Kunde)

    Public Property Kunden As ObservableCollection(Of Kunde)
        Get
            Return _Kunden
        End Get
        Set(value As ObservableCollection(Of Kunde))
```

```
            _Kunden = value
        End Set
    End Property

    Public Sub New()
        InitializeComponent()
        dbContext = New BestellverwaltungContext
        Kunden = New ObservableCollection(Of Kunde)(dbContext.Kunden)
        DataContext = Me
    End Sub
End Class
```

Den XAML-Code haben wir so einfach wie möglich gehalten:

```
<Window x:Class="MainWindow" ... Title="MainWindow" Height="450" Width="800">
    <Grid>
        <DataGrid ItemsSource="{Binding Kunden}"></DataGrid>
    </Grid>
</Window>
```

Die Anweisung der Konstruktor-Methode **New()**, die wir als Ausgangspunkt für die folgenden Beispiele nutzen, ist diese:

```
Kunden = New ObservableCollection(Of Kunde)(dbContext.Kunden)
```

Hier weisen wir der **ObservableCollection** mit Elementen des Typs **Kunde** einfach alle Kunden des Datenbankkontextes zu, also **dbContext.Kunden**. Statt **dbContext.Kunden** können wir auch die nachfolgend vorgestellten Abfrageausdrücke verwenden.

Einfache Auswahlabfrage

Die einfachste Syntax bei einer LINQ-Abfrage sieht wie folgt aus – hier noch in Zusammenhang mit unserer Anweisung:

```
Kunden = New ObservableCollection(Of Kunde)(From Kunde In dbContext.Kunden)
```

Der Einfachheit halber schauen wir uns nun noch den Teil an, den wir in eine **ObservableCollection** umwandeln, hier also diesen Teil:

```
From Kunde In dbContext.Kunden
```

Dies liefert alle Elemente der **Kunden**-Auflistung. **dbContext.Kunden** ist dabei die Quelle der Daten. **Kunde** ist eine Variable, auf die wir mit weiteren Schlüsselwörtern der Abfragesprache LINQ zugreifen können. Diese Abfrage liefert das gleiche Ergebnis wie **dbKontext.Kunden**.

Eine ausführlichere Schreibweise unter Einbeziehung des **Select**-Schlüsselworts sieht wie folgt aus und liefert wiederum das gleiche Ergebnis:

```
From Kunde In dbContext.Kunden Select Kunde
```

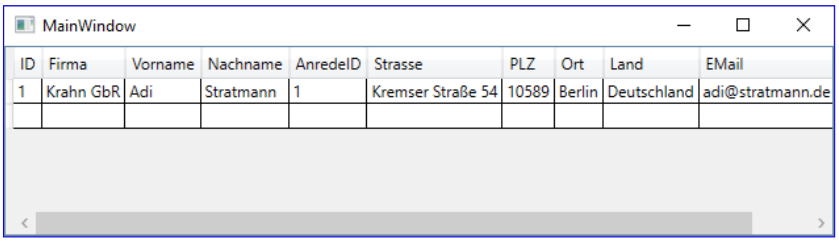


Bild 2: DataGrid mit dem Kunden mit dem Wert **1** im Feld **ID**

Daten filtern mit Where

Wenn Ihnen die bisherigen Schlüsselwörter schon von SQL bekannt vorkamen, gibt es gute Nachrichten – das **Where**-Schlüsselwort dürften Sie auch kennen. Dieses hängen Sie hinter die **From**-Anweisung an und vor der **Select**-Anweisung.

Das folgende Beispiel selektiert beispielsweise nur den Kunden, dessen Feld **ID** den Wert **1** aufweist (Ergebnis siehe Bild 2):

```
From Kunde In dbContext.Kunden Where Kunde.ID = 1 Select Kunde
```

Ersatz für den SQL-LIKE-Operator: Contains, StartsWith und EndsWith

Unter SQL haben Sie Vergleiche mit Platzhaltern wie dem Sternchen mit dem **LIKE**-Operator realisiert, also etwa mit **SELECT * FROM tblKunden WHERE PLZ LIKE "1*"**, um alle Kunden mit einer PLZ, die mit **1** beginnt, zu ermitteln.

Unter LINQ gibt es dazu die drei Operatoren **Contains**, **StartsWith** und **EndsWith**:

- **Contains**: Entspricht der Suche nach einem Ausdruck, der irgendwo im Feldinhalt vorkommen darf, also gleichbedeutend mit **LIKE "*<Suchbegriff>*"**.
- **StartsWith**: Sucht nach Datensätzen, deren zu durchsuchendes Feld mit dem angegebenen Ausdruck beginnt. Unter SQL wäre das **LIKE "<Suchbegriff>*"**.
- **EndsWith**: Sucht nach Datensätzen, deren zu durchsuchendes Feld auf den angegebenen Ausdruck endet. Unter SQL entspricht das **LIKE "*<Suchbegriff>"**.

Die Operatoren hängen wir an das zu durchsuchende Feld an und geben den Suchbegriff in Klammern an. Alle Kunden, deren **PLZ** mit **1** beginnt, ermitteln wir also mit dem folgenden Ausdruck:

```
From Kunde In dbContext.Kunden Where Kunde.PLZ.StartsWith("1") Select Kunde
```

Kriterien verknüpfen

Die Suchkriterien verknüpfen Sie einfach wie von SQL gewohnt mit den Operatoren **Or** und **And**. Kunden, deren Vorname und Nachname mit **A** beginnt, finden Sie beispielsweise so:

```
From Kunde In dbContext.Kunden Where Kunde.Vorname.StartsWith("A") Or Kunde.Nachname.StartsWith("A") Select Kunde
```

Daten sortieren mit Order By

Natürlich können Sie die Daten auch in beliebiger Sortierung in die **ObservableCollection** füllen. Dazu verwenden wir den folgenden Abfrageausdruck, in dem wir das **Order By**-Schlüsselwort gefolgt von dem zu sortierenden Feld angeben:

```
From Kunde In dbContext.Kunden Order By Kunde.Nachname Select Kunde
```

Wenn Sie nach mehreren Feldern sortieren wollen, geben Sie diese durch ein Komma voneinander getrennt hinter dem **Order By**-Schlüsselwort an:

```
From Kunde In dbContext.Kunden Order By Kunde.Nachname, Kunde.Vorname Select Kunde
```

Hier haben wir noch nicht explizit die Sortierreihenfolge angegeben. In diesem Fall sortiert die Abfrage aufsteigend nach den angegebenen Feldern. Sie können die Sortierreihenfolge auch explizit angeben, indem Sie eines der Schlüsselwörter **Ascending** (**Aufsteigend**) oder **Descending** (**Absteigend**) hinter dem Sortierfeld angeben:

```
From Kunde In dbContext.Kunden Order By Kunde.Nachname Ascending, Kunde.Vorname Descending Select Kunde
```

Reihenfolge der Elemente

Im Gegensatz zu SQL, wo die Reihenfolge strikt festgelegt ist (zum Beispiel **SELECT ... FROM ... WHERE ... ORDER BY**), können Sie in LINQ etwa die Reihenfolge der **Where**- und der **Order By**-Klausel vertauschen. Wir empfehlen jedoch, die Reihenfolge wie unter SQL zu gestalten, um die Lesbarkeit zu erhöhen. Die folgende Abfrage, bei der wir die **Order By**-Klausel vor der **Where**-Klausel platziert haben, funktioniert jedoch genauso wie die mit der umgekehrten Reihenfolge:

```
From Kunde In dbContext.Kunden Order By Kunde.Nachname Ascending Where Kunde.Nachname.StartsWith("A") Select Kunde
```

Die »Laufvariable«

Die Bezeichnung zwischen **From** und **In**, in unserem Fall also etwa **Kunde**, können Sie beliebig wählen. Wenn Sie den Abfragecode kürzer darstellen wollen, können Sie auch einfach **k** verwenden:

```
From k In dbContext.Kunden Where k.Nachname.StartsWith("A") Order By k.Nachname Ascending Select k
```

Zu selektierende Daten festlegen mit Select

Die **Select**-Anweisung haben wir in den obigen Beispielen schon ausgiebig genutzt. Allerdings haben wir mit **... Select Kunde** immer einfach die kompletten **Kunde**-Objekte selektiert. Das hätten wir auch erreicht, wenn wir die **Select**-Klausel weggelassen hätten. **Select** bietet allerdings eine Menge mehr als nur die Möglichkeit, genau die Objekte zurückzuliefern, welche die zu untersuchende Auflistung bietet, in diesem Fall die Liste der **Kunde**-Objekte. In allen Fällen, in denen Sie nicht einfach die Objekte zurückliefern, die in der zu untersuchenden Auflistung enthalten sind, handelt es sich um eine sogenannte Projektion. Schauen wir uns an, wie das aussehen kann! Wenn wir beispielsweise nur die ID, den Vornamen und den Nachnamen des Kunden erhalten wollen, formulieren wir den Ausdruck wie folgt:

```
From k In dbContext.Kunden Select k.ID, k.Vorname, k.Nachname
```

Allerdings liefert dies dann kein Objekt vom Typ **Kunde** mehr zurück, was sich in einer entsprechenden Fehlermeldung bemerkbar macht (siehe Bild 3). Also spielen wir erst einmal außerhalb des DataGrids mit dieser Möglichkeit:

```
Dim KundenMitName = From k In dbContext.Kunden Select k.ID, k.Vorname, k.Nachname
For Each k In KundenMitName
    Debug.Print(k.ID.ToString() + " " + k.Vorname + " " + k.Nachname)
Next
```

Wir weisen also das Ergebnis, das die Eigenschaften **ID**, **Vorname** und **Nachname** der Elemente aus **dbContext.Kunden** enthält, der Variablen **KundenMitName** zu. **KundenMitName** wir dann zu einer Variablen des Typs **DbQuery**. Auf die enthaltenen Eigenschaften können wir wie auf die Eigenschaften einer Klasse zugreifen. Im Beispiel oben durchlaufen wir diese Elemente in einer **For Each**-Schleife, wobei wir das aktuelle Element der Variablen **k** zuweisen. Diese hält dann die Eigenschaften **ID**, **Vorname** und **Nachname** bereit. Diese geben wir innerhalb der Schleife mit der **Debug.Print**-Anweisung im **Output**-Fenster aus.

DbQuery-Ergebnis in DataGrid anzeigen

Wie aber zeigen wir dieses Ergebnis im DataGrid an? Dieses ist ja an die Eigenschaft **Kunden** der Code behind-Klasse gebunden. Wenn wir dies nicht ändern wollen, müssen wir die Informationen **ID**, **Vorname** und **Nachname** anderweitig in Elemente des Typs **Kunde** schreiben und diese dann der Auflistung zuweisen. Aber gelingt uns das? Der theoretische Ansatz sieht wie folgt aus:

```
Dim KundenMitName = From k In dbContext.Kunden Select New Kunde With {.ID = k.ID, .Vorname = k.Vorname, .Nachname =
k.Nachname}
Kunden = New ObservableCollection(Of Kunde)(KundenMitName)
```

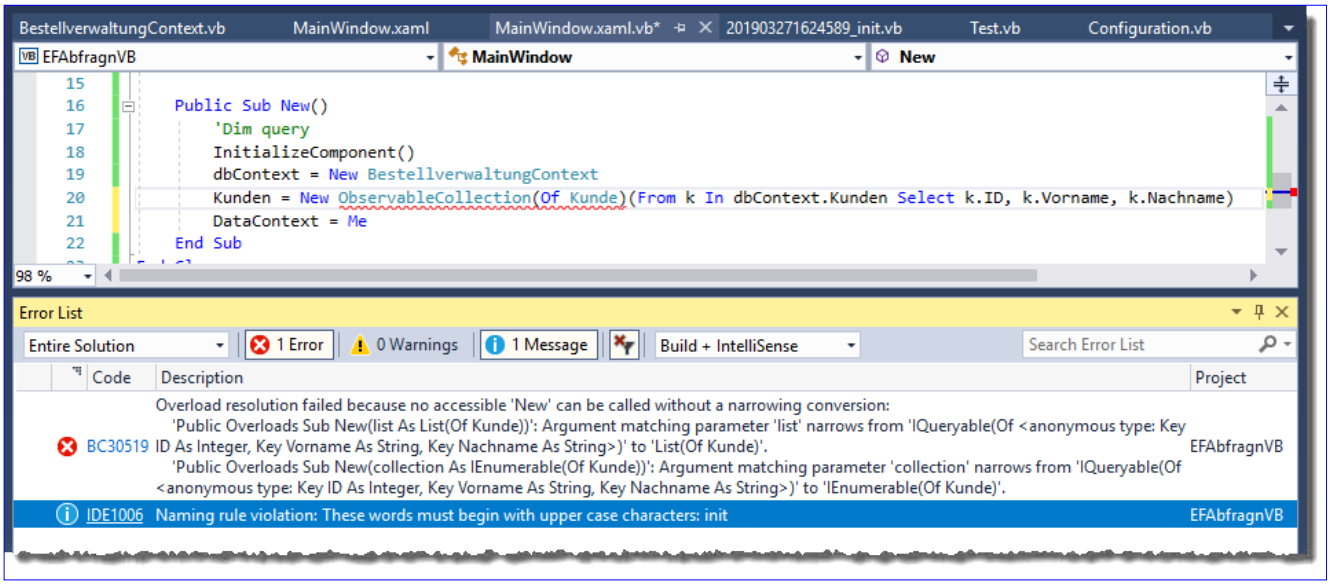


Bild 3: Fehlermeldung, wenn keine Objekte des Typs **Kunde** zurückgeliefert werden

Hier weisen wir einer Variablen namens **KundenMitName** das Ergebnis der Abfrage in Form von Objekten mit den Eigenschaften **ID**, **Vorname** und **Nachname** zu. Danach wollen wir das Ergebnis der öffentlichen **ObservableCollection**-Variablen **Kunden** zuweisen. Das liefert allerdings den Fehler **The entity or complex type 'EFAbfragnVB.Kunde' cannot be constructed in a LINQ to Entities query**. Der Grund für den Fehler ist, dass die Kunde-Entitäten nur teilweise mit den Daten aus der Datenbank gefüllt werden, was in einem nicht aktualisierbaren Zustand resultiert.

Also verwenden wir doch eine neue Klasse, um das neue Ergebnis zu speichern. Diese Klasse sieht wie folgt aus:

```
Public Class KundeMitName
    Public Property ID As System.Int32
    <StringLength(255)>
    <Required>
    Public Property Vorname As System.String
    <StringLength(255)>
    <Required>
    Public Property Nachname As System.String
End Class
```

Den Code der Code behind-Klasse von **KundenMitName.xaml** gestalten wir wie folgt, wobei wir die neue private Variable und die öffentliche Eigenschaft **_KundenMitName** und **KundenMitName** nennen:

```
Public Class KundenMitName
    Private dbContext As BestellverwaltungContext
    Private _KundenMitName As List(Of KundeMitName)

    Public Property KundenMitName As List(Of KundeMitName)
        Get
            Return _KundenMitName
        End Get
        Set(value As List(Of KundeMitName))
            _KundenMitName = value
        End Set
    End Property
```

In der Konstruktor-Methode füllen wir die Variable **KundenMitNamen** dann über die folgende Anweisung, in der wir hinter der **Select**-Klausel jeweils ein neues **KundeMitName**-Objekt erzeugen und seinen drei Eigenschaften **ID**, **Vorname** und **Nachname** die entsprechenden Werte des jeweiligen Datensatzes zuweisen:

```
    Public Sub New()
        InitializeComponent()
        dbContext = New BestellverwaltungContext
```

Entity Framework: Gespeicherte Prozeduren

Entity Framework bietet die Möglichkeit, mit einer Datenbank zu arbeiten, die lediglich einfache Tabellen enthält. Sie könnten so sämtliche Geschäftslogik in der Anwendung halten. Manch einer mag aber vielleicht Teile der Geschäftslogik in die Datenbank überführen oder, wenn die Datenbank schon existiert, dort belassen, damit diese von verschiedenen Anwendungen aus genutzt werden kann. Ein Beispiel sind Trigger, die bei Datenänderungen automatisch ausgelöst werden, ein anderes sind gespeicherte Prozeduren, über die sie sowohl Daten abfragen als auch anlegen, ändern oder löschen können. Dieser Artikel beschäftigt sich mit den gespeicherten Prozeduren und den Möglichkeiten, die sich unter Entity Framework zu ihrer Nutzung bieten.

Wenn Sie ein Objekt in Entity Framework anlegen wollen, erstellen Sie das Objekt neu und rufen die **Add**-Methode des jeweiligen DbSets auf. Der anschließende Aufruf der **SaveChanges**-Methode sorgt dann für das Übertragen der Änderung in der lokalen Auflistung in die Datenbank. Wenn Sie Eigenschaften eines Objekts ändern, übertragen Sie die Änderungen ebenfalls mit der **SaveChanges**-Methode. Und das geschieht auch, wenn Sie ein Element etwa mit der **Remove**-Methode entfernt haben – erst die **SaveChanges**-Methode löscht den entsprechenden Datensatz aus der zugrundeliegenden Tabelle.

Ausführung gespeicherte Prozeduren unter LINQ beobachten

Im Hintergrund führt die Anwendung entsprechende SQL-Anweisungen aus, wovon wir uns leicht überzeugen können, wenn wir den SQL Server Profiler mitlaufen lassen. Die grundlegende Bedienung erläutern wir im Artikel **SQL Server-Interaktion mit dem Profiler verfolgen** in Ausgabe 4/2016.

In unserem Fall wollen wir für die aktuelle Datenbank, deren ID wir im SQL Server Management Studio mit der folgenden Abfrage für die Zieldatenbank ermitteln, die ausgeführten SQL-Anweisungen anzeigen lassen:

```
SELECT DB_ID()
```

Das Ergebnis lautet beispielsweise **40**. In den Ablaufverfolgungseigenschaften des SQL Server Profilers aktivieren Sie die Option **Alle Spalten anzeigen**. Dann können Sie nach einem Klick auf **Spaltenfilter** im Dialog **Filter bearbeiten** einen Filter für die Eigenschaft **DatabaseID** festlegen, und zwar auf den oben ermitteln Wert, in unserem Fall **40** (siehe Bild 1).

Danach aktivieren Sie noch die Option **Alle Ereignisse anzeigen** und selektieren dann im Bereich **Ereignisauswahl** das Ereignis **Stored Procedures\RCP: Completed** (siehe Bild 2). Damit legen wir fest, dass alle per Remote ausgeführten Abfragen vom Profiler erfasst und ausgegeben werden.

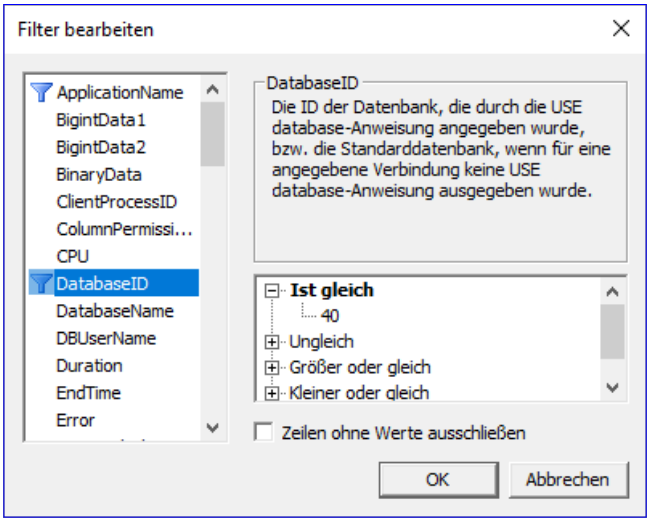


Bild 1: Festlegen eines Spaltenfilters

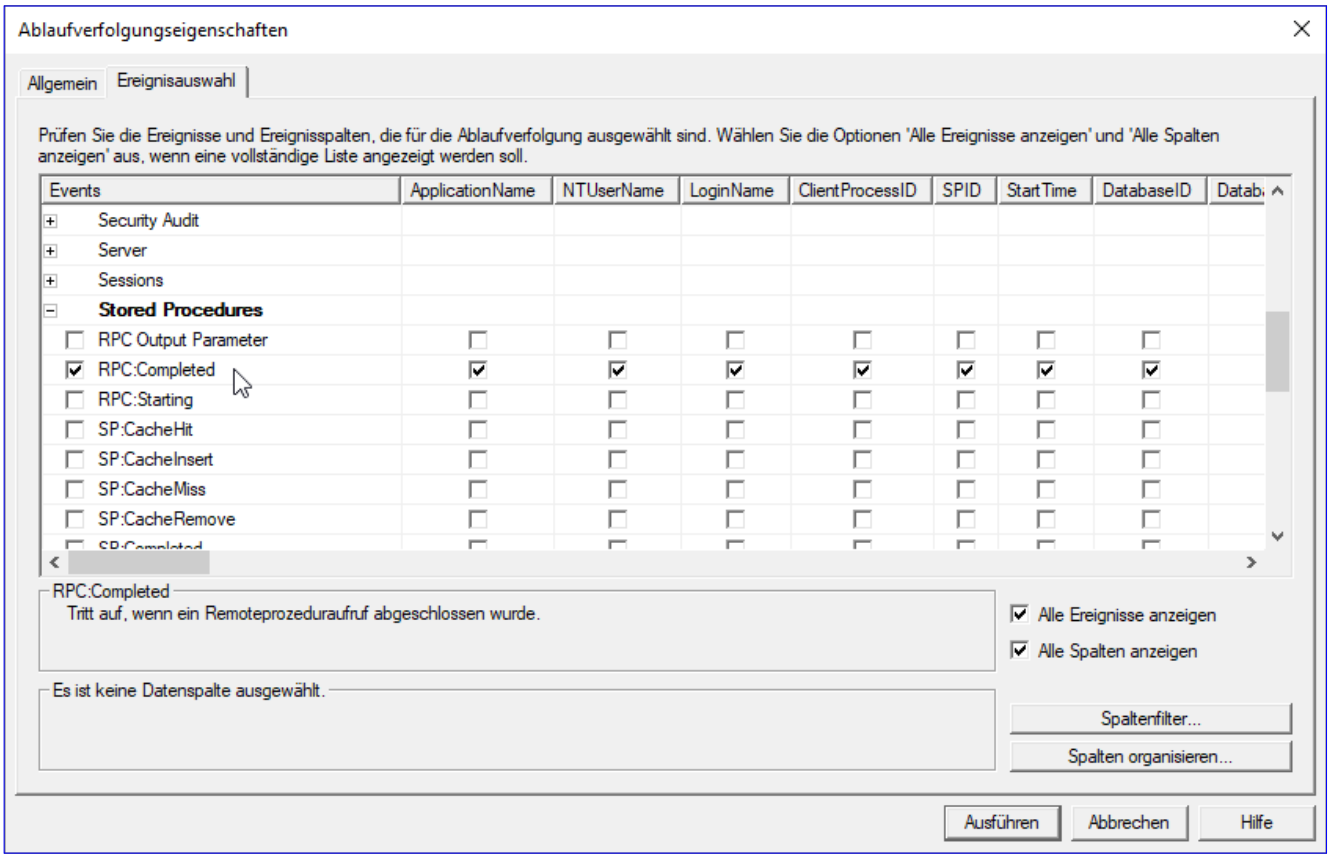


Bild 2: Auswahl der zu filternden SQL-Aktionen

Beispielprojekt vorbereiten

Als Beispielprojekt verwenden wir ein Projekt auf Basis der Vorlage **Visual BasicIWindows DesktopIWPF App**. Dieser fügen wir mit den Methoden aus dem Artikel **Von Access zu EF: Step by step** aus Ausgabe 5/2018, erweitert um die Informationen aus **Von Access zu EF: Update 1** aus der vorliegenden Ausgabe, ein Entity Data Model hinzu, aus dem wir dann eine entsprechende SQL Server-Datenbank generieren.

Automatisch generierte gespeicherte Prozedur ermitteln

Um zu erkennen, welche gespeicherten Prozeduren im Hintergrund ausgeführt werden, wenn wir Daten ändern, löschen oder hinzufügen, legen wir für das Fenster **MainWindow.xaml** einen Konstruktor in der Code behind-Klasse an, den wir wie folgt ausstatten.

Wir deklarieren dort einen Datenbankkontext namens **dbContext** und füllen diesen mit einer neuen Instanz unserer Datenbankkontext-Klasse **BestellverwaltungContext**:

```
Public Sub New()
    Dim dbContext As BestellverwaltungContext
    Dim Kunde As Kunde
    dbContext = New BestellverwaltungContext
```

Dann weisen wir der Objektvariablen **Kunde** den ersten Kunden zu, den wir in der Tabelle **Kunden** finden, bearbeiten die Eigenschaft **Firma** des Kunden und speichern die Änderung durch den Aufruf der Methode **SaveChanges**:

```
Kunde = dbContext.Kunden.First
Kunde.Firma = Kunde.Firma + "1"
dbContext.SaveChanges()
```

Dann entfernen wir diesen Kunden und speichern die Änderungen ebenfalls mit **SaveChanges**:

```
dbContext.Kunden.Remove(Kunde)
dbContext.SaveChanges()
```

Schließlich legen wir noch einen neuen Kunden an, fügen das **Kunde**-Element zur Auflistung der Kunden mit der **Add**-Methode hinzu und speichern die Änderungen mit **SaveChanges**:

```
    Kunde = New Kunde With {.Firma = "Firma", .AnredeID = 1, .Vorname = "Klaus", .Nachname = "Müller", _
        .EMail = "klaus@mueller.de", .Land = "Deutschland", .Ort = "Testhausen", .PLZ = "11111", .Strasse = "Teststr. 1"}
    dbContext.Kunden.Add(Kunde)
    dbContext.SaveChanges()
End Sub
```

Bevor wir die Anwendung öffnen, starten wir den SQL Server Profiler mit den obigen Einstellungen. Setzen Sie einen Haltepunkt in die erste Zeile mit der Anweisung **dbContext.SaveChanges**. Dann starten Sie die Anwendung. Nachdem der Haltepunkt erreicht ist, lassen Sie durch Betätigen der Taste **F11** die **SaveChanges**-Methode ausführen und schauen sich das Ergebnis im SQL Server Profiler an. Hier finden Sie die SQL-Anweisung mit dem **UPDATE**-Schlüsselwort aus Bild 3 vor.

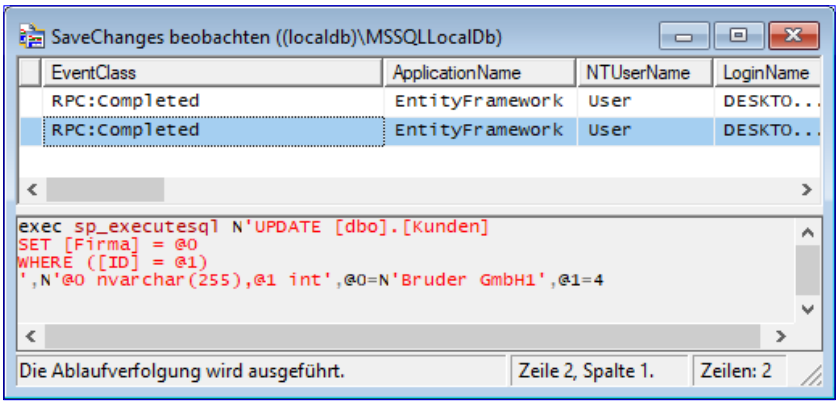


Bild 3: SQL-Anweisung zum Ändern eines Datensatzes

Die gespeicherte Prozedur, die für den zweiten **SaveChanges**-Aufruf nach dem Entfernen eines der Elemente ausgeführt wird, lautet wie folgt und enthält erwartungsgemäß das **DELETE**-Schlüsselwort:

```
exec sp_executesql N'DELETE [dbo].[Kunden]
WHERE ([ID] = @0)',N'@0 int',@0=4
```

Fehlt noch die letzte SQL-Anweisung zum Anlegen eines Datensatzes nach dem Hinzufügen eines neuen **Kunde**-Objekts:

Entity Framework: Daten archivieren

Im Artikel »Entity Framework: Gespeicherte Prozeduren« haben wir gezeigt, wie Sie gespeicherte Prozeduren für die Entitäten einer Anwendung anlegen und diese beim Ändern, Anlegen oder Löschen von Entitäten nutzen. Damit diese gespeicherten Prozeduren auch einen Mehrwert haben, zeigen wir im vorliegenden Artikel, wie Sie die gespeicherten Prozeduren so erweitern, dass Sie etwa vor dem Ändern eines Datensatzes eine Kopie dieses Datensatzes in einer für diesen Zweck vorgesehenen Tabelle anlegen, um später die Änderungen reproduzieren und wiederherstellen zu können.

Voraussetzung

Wir starten in diesem Artikel mit dem Beispielprojekt, das wir im Beitrag **Entity Framework: Gespeicherte Prozeduren** erstellt haben. Dieses enthält bereits die Entitäten unserer Bestellverwaltung und die Migrationsklassen, mit denen wir die Datenbank auf Basis der Entitäten erstellen und über die **Seed**-Methode mit Daten füllen können. Die Aufgabe ist nun, erstens eine Tabelle namens **KundenArchiv** zu erstellen, in der wir die alten Kundendaten speichern können. Dabei wollen wir den verwendeten Primärschlüsselwert des Datensatzes sichern sowie das Datum der Änderung. Danach wollen wir die gespeicherte Prozedur **dbo.Kunden_Update** so anpassen, dass vor dem Durchführen der Änderungen am Datensatz die alte Version des Datensatzes in der Tabelle **dbo.Kunden_Update** gespeichert wird. Außerdem soll auch die Tabelle zum Archivieren der Daten per Update-Database erstellt werden, sodass wir entsprechenden Code in einer Migrationsklasse anlegen müssen.

Gespeicherte Prozeduren anpassen

Wir wollen die gespeicherten Prozedur **dbo.Kunden_Update** so anpassen, dass diese in einer noch zu erstellenden Tabelle namens **KundenArchiv** die Version der Datensätze der Tabelle **Kunden** vor der Änderung speichert. Dazu benötigen wir zunächst eine solche Tabelle. Diese legen wir mit dem folgenden Skript an, das wir in einem neuen Abfragefenster für die aktuelle Datenbank ausführen. Das Abfragefenster öffnen wir, indem wir im SQL Server-Objekt-Explorer mit der rechten Maustaste auf den Eintrag für die Datenbank klicken und den Eintrag **New Query...** (deutsch **Neue Abfrage...**) betätigen. Da wir nur wenige Änderungen gegenüber der Originaltabelle **Kunden** durchführen wollen, können wir den Code zur Erstellung der Tabelle **Kunden** als Vorlage verwenden. Diesen erhalten Sie, indem Sie mit der rechten Maustaste auf die Tabelle **Kunden** klicken und aus dem Kontextmenü den Eintrag **Skripterstellung als...lCREATE inlNeues Abfragefenster** auswählen. Hier ändern Sie den Tabellennamen **Kunden** überall in **KundenArchiv**. Außerdem fügen wir das Feld **GeaendertAm** hinzu, welches das Änderungsdatum speichern soll. Das zweite neue Feld heißt **OriginalID** und soll die ID des Datensatzes enthalten, der geändert wurde:

```
CREATE TABLE [dbo].[KundenArchiv] (
    [ID]          INT            IDENTITY (1, 1) NOT NULL,
    [OriginalID]  INT            NOT NULL,
    [Firma]       NVARCHAR (255) NULL,
    [Vorname]     NVARCHAR (255) NOT NULL,
    [Nachname]    NVARCHAR (255) NOT NULL,
    [AnredeID]    INT            NOT NULL,
    [Strasse]     NVARCHAR (255) NOT NULL,
```

```
    [PLZ]         NVARCHAR (255) NOT NULL,
    [Ort]         NVARCHAR (255) NOT NULL,
    [Land]        NVARCHAR (255) NOT NULL,
    [EMail]       NVARCHAR (255) NOT NULL,
    [GeaendertAm] DATETIME       NOT NULL
);
GO
CREATE NONCLUSTERED INDEX [IX_AnredeID]
    ON [dbo].[KundenArchiv]([AnredeID] ASC);
GO
ALTER TABLE [dbo].[KundenArchiv]
    ADD CONSTRAINT [PK_dbo.KundenArchiv] PRIMARY KEY CLUSTERED ([ID] ASC);
GO
ALTER TABLE [dbo].[KundenArchiv]
    ADD CONSTRAINT [FK_dbo.KundenArchiv_dbo.Anreden_AnredeID] FOREIGN KEY ([AnredeID]) REFERENCES [dbo].[Anreden] ([ID])
ON DELETE CASCADE;
```

Führen Sie dieses Skript aus, indem Sie auf die **Starten**-Schaltfläche oben im Skriptfenster klicken (nicht zu verwechseln mit der **Starten**-Schaltfläche für das Projekt in der Symbolleiste von Visual Studio). Nach der Aktualisierung des Elements **Tabellen** im SQL Server-Objekt-Explorer erscheint dort die neue Tabelle **KundenArchiv**.

Gespeicherte Prozedur »Kunden_Update« anpassen

Nun müssen wir die gespeicherte Prozedur **dbo.Kunden_Update** so anpassen, dass nicht nur die Änderungen am aktuellen Datensatz gespeichert werden, sondern zuvor noch die aktuelle Version des Datensatzes in der Tabelle **KundenArchiv** archiviert wird. Diese Änderung könnten Sie zur Migration namens **StoredProcedures** hinzufügen, sofern Sie diese noch nicht ausgeführt haben. Allerdings können wir auch eine neue Migration anlegen und dort unseren Code einfügen. Dazu geben Sie einfach den folgenden Befehl in die Paketmanager-Konsole ein:

```
add-migration AlterStoredProcedure
```

Es erscheint eine leere Klasse. Diese hat wieder zwei Methoden – die **Up**- und die **Down**-Methode. Nochmal zur Verdeutlichung: Wenn Sie in der Paketmanager-Konsole **Update-Database** aufrufen, werden alle **Up**-Methoden der vorhandenen Migrationsklassen aufgerufen, die noch nicht ausgeführt wurden. Deshalb fügen wir den Aufruf der Methode **AlterStoredProcedure**, die zum Ändern einer gespeicherten Prozedur zum Einsatz kommt, für die **Up**-Methode ein:

```
Public Partial Class AlterStoredProcedure
    Inherits DbMigration

    Public Overrides Sub Up()
        AlterStoredProcedure(
            "dbo.Kunde_Update",
```

```
        Function(p) New With
            {
                .ID = p.Int(), .Firma = p.String(maxLength:=255), .Vorname = p.String(maxLength:=255),
                .Nachname = p.String(maxLength:=255), .AnredeID = p.Int(), .Strasse = p.String(maxLength:=255),
                .PLZ = p.String(maxLength:=255), .Ort = p.String(maxLength:=255), .Land = p.String(maxLength:=255),
                .EMail = p.String(maxLength:=255)
            },
        body:="INSERT [dbo].[KundenArchiv]([OriginalID], [Firma], [Vorname], [Nachname], [AnredeID], [Strasse], "
            & "[PLZ], [Ort], [Land], [EMail], [GeaendertAm])" & vbCrLf &
            "SELECT ID, Firma, Vorname, Nachname, AnredeID, Strasse, PLZ, Ort, Land, EMail, "
            & "GETDATE() AS GeaendertAm FROM Kunden WHERE ID = @ID;" & vbCrLf &
            "UPDATE [dbo].[Kunden]" & vbCrLf &
            "SET [Firma] = @Firma, [Vorname] = @Vorname, [Nachname] = @Nachname, [AnredeID] = @AnredeID, " &
            "[Strasse] = @Strasse, [PLZ] = @PLZ, [Ort] = @Ort, [Land] = @Land, [EMail] = @EMail" & vbCrLf &
            "WHERE ([ID] = @ID);"
    )
End Sub
```

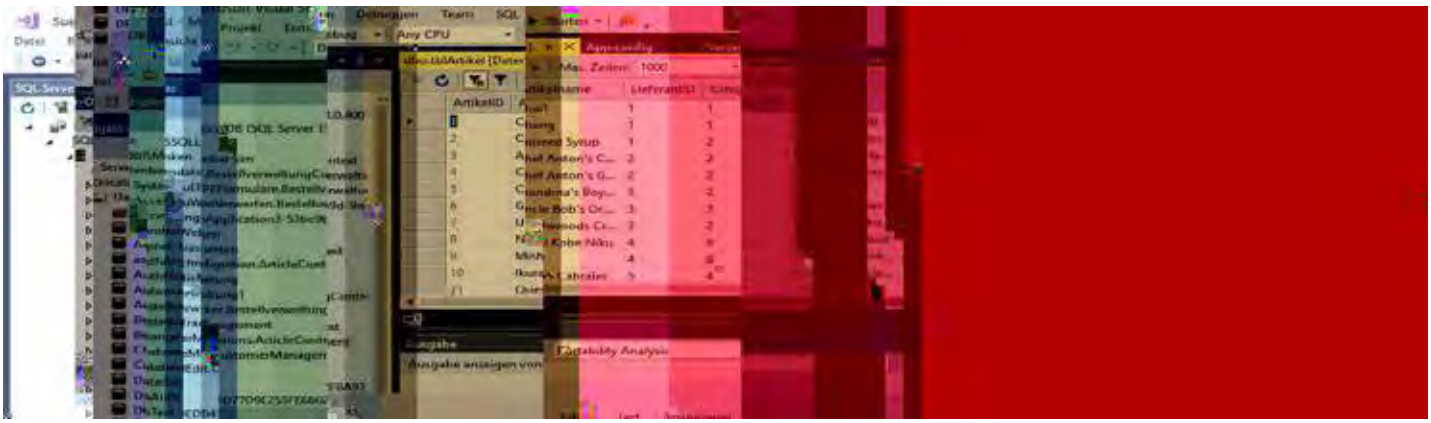
Im Gegensatz zur vorherigen Version der gespeicherten Prozedur **dbo.Kunde_Update** hat diese noch eine vorgeschaltete **INSERT INTO**-Anweisung, welche die aktuelle Version des gespeicherten und zu ändernden Datensatzes in die Tabelle **Kunden-Archiv** sichert.

Die **Down**-Methode wird immer ausgeführt, wenn Sie eine Migration zurücknehmen. Wenn also etwa die vor der aktuellen Migration ausgeführte Migration wie in unserem Beispiel **StoredProcedures**, dann können Sie mit der Anweisung **update-database -targetmigration:"StoredProcedures"** wieder den Stand der vorherigen Migration herstellen. Dazu müssen Sie natürlich die **Down**-Methode so programmieren, dass dies auch gelingt. In unserem Fall rufen wir die **AlterStoredProcedure**-Anweisung auf und ändern die gespeicherte Prozedur **dbo.Kunde_Update** wieder auf den Stand der Migration **StoredProcedures**:

AUSGABE 02-03/2019

DATENBANK ENTWICKLER

MAGAZIN FÜR DIE DATENBANKENTWICKLUNG MIT VISUAL STUDIO FÜR DESKTOP, WEB UND CO.



TOP-THEMEN:

STEUERELEMENTE	Mehrspaltige Kombinationsfelder	SEITE 14
USERINTERFACE	Entity Framework: Bilder in WPF	SEITE 23
EF	Blättern in Datensätzen	SEITE 53
EF	Änderungen erkennen und verwerfen	SEITE 88
ACCESS ZU .NET	Detailformulare mit Combo, Checkbox und Button	SEITE 96

Bytes im Griff mit der Stream-Klasse: FileStream

Die Stream-Klasse ist die Basisklasse für verschiedene Stream-Klassen wie FileStream oder MemoryStream. Streams bieten die Möglichkeit, Byte-genau schreibend wie lesend auf Daten aus Dateien, dem Arbeitsspeicher oder dem Netzwerk zuzugreifen. Dieser Artikel zeigt, welche Stream-Klassen es gibt und geht genauer auf die FileStream-Klasse ein. Dabei zeigen wir, wie Sie mit dieser Klasse lesend und schreibend auf den Inhalt von Dateien zugreifen und den Inhalt mit Byte-Arrays austauschen.

Verschiedene Stream-Klassen

Es gibt folgende von der **Stream**-Klasse abgeleitete Klassen:

- **BufferedStream**: Puffert Daten aus E/A-Datenströmen im Arbeitsspeicher.
- **CryptoStream**: Verschlüsselt Daten.
- **FileStream**: Schreibt Daten in Dateien im Dateisystem.
- **GZipStream**: Komprimiert und dekomprimiert Streams.
- **MemoryStream**: Schreibt einen Stream in den Hauptspeicher statt in eine temporäre Datei.
- **NetworkStream**: Erlaubt den Zugriff auf Netzwerkressourcen.

Wir wollen uns in diesem Artikel auf die **FileStream**- und die **MemoryStream**-Klasse konzentrieren.

Die FileStream-Klasse

Mit der **FileStream**-Klasse können Sie byteweise aus Dateien lesen und in Dateien schreiben sowie einen Positionszeiger in einen Stream setzen. Der FileStream puffert dabei die Daten (bis zu acht Kilobyte). Wenn Sie ein Objekt auf Basis der **FileStream**-Klasse erzeugen wollen, können Sie per Konstruktor verschiedene Parameter übergeben:

- **path**: Pfad der zu lesenden oder schreibenden Datei
- **mode**: Modus für den Dateizugriff. Die Enumeration **FileMode** bietet folgende Werte: **Append**, **Create**, **CreateNew**, **Open**, **OpenOrCreate**, **Truncate**
- **access**: Art des Dateizugriffs mit folgenden Werten der Enumeration **FileAccess**: **Read**, **ReadWrite**, **Write**
- **share**: Gibt an, ob gemeinsamer Zugriff auf die Datei möglich ist. Die Enumeration **FileShare** liefert dazu die folgenden möglichen Werte: **Delete**, **Inheritable**, **None**, **Read**, **ReadWrite**, **Write**

- **bufferSize**: Ein **int32**-Wert größer, der die Buffer-Größe angibt. Standardwert ist 4.096.

- **isAsync**: **Boolean**-Wert, der angibt, ob asynchrone Zugriffe möglich sein sollen.

Byte-Array in eine Datei schreiben

Die folgenden Anweisungen schreiben ein Byte-Array in eine neue Datei. Zuerst deklarieren wir die nötigen Elemente, legen den Pfad fest und erstellen das Array mit Zahlenwerten:

```
Dim objFileStream As FileStream
Dim strPath As String
Dim byt As Byte()
strPath = "C:\...\Stream.txt"
byt = {1, 2, 3, 4, 5, 6, 7, 8, 9, 10}
```

Dann erstellen wir das **FileStream**-Objekt mit dem Pfad und mit dem Modus **Create**, was bedeutet, dass die Datei neu erstellt wird, auch wenn sie bereits existiert:

```
objFileStream = New FileStream(strPath, FileMode.Create)
```

Schließlich schreiben wir mit der **Write**-Methode den Inhalt des Arrays ab der Position **0** der Datei bis zur Position, die durch die Länge des Arrays angegeben wird, und schließen die Datei wieder:

```
With objFileStream
    .Write(byt, 0, byt.Length)
    .Close
End With
```

Byte-Array aus einer Datei lesen

Die folgenden Codezeilen erlauben das Lesen eines Byte-Arrays aus einer Datei. Auch hier deklarieren wir wieder einige Variablen, wobei wir hier direkt die Größe des Byte-Arrays angeben:

```
Dim objFileStream As FileStream
Dim strPath As String
Dim byt(9) As Byte
strPath = "C:\Users\User\Dropbox\Daten\Fachmagazine\VisualStudioDatenbankentwicklung\2019\02\Streams\Stream.txt"
Dim l As Long
```

Das **FileStream**-Objekt erstellen wir diesmal mit dem Wert **FileMode.Open** als zweiten Parameter.

Damit öffnen wir die Datei zum Lesen. Dann greifen wir mit der **Read**-Methode auf die Datei zu und lesen in das Array **byt** von der ersten Position mit der Länge des **FileStream**-Objekts ein:

WPF: DataTrigger und MultiDataTrigger

Um die Eigenschaftswerte von Steuerelementen abhängig von den Werten anderer Steuerelemente oder auch von Eigenschaften einzustellen, verwenden Sie DataTrigger oder MultiDataTrigger. Wir schauen uns in diesem Artikel die Grundlagen zu diesen Elementen an und zeigen anhand einiger Beispiele, was beim Praxiseinsatz zu beachten ist. Dabei betrachten wir auch, wie Sie DataTrigger auf Basis von Eigenschaften im Code behind-Modul des WPF-Objekts verwenden können.

Eigenschaften per VB einstellen

Wer von Access kommt, kennt nur zwei Methoden, um die Eigenschaften eines Steuerelements einzustellen: über das Eigenschaftsfenster oder per VBA. Beide Methoden sind beliebig miteinander kombinierbar. Sie können einen initialen Wert über das Eigenschaftsfenster einstellen oder dies per VBA erledigen, etwa in der Ereignisprozedur **Form_Load** des Formulars. Nachträgliche Einstellungen sind allerdings nur per VBA möglich.

Unter WPF/VB gibt es umfangreichere Möglichkeiten. Sie können eine Eigenschaft natürlich explizit im WPF-Code eines Elements festlegen – um etwa ein Steuerelement gleich beim Anzeigen des Fensters zu deaktivieren, stellen Sie die Eigenschaft einfach wie folgt ein (siehe Fenster **EigenschaftPerWPF.xaml**):

```
<Button x:Name="btnBeimAnzeigenDeaktiviert" ... IsEnabled="False" />
```

Sie können dem Fenster dann zwei Schaltflächen hinzufügen, mit deren Ereignismethoden für das Attribut **Click** die Eigenschaft **IsEnabled** auf **True** oder **False** eingestellt wird:

```
Private Sub btnAktivieren_Click(sender As Object, e As RoutedEventArgs)
    btnBeimAnzeigenDeaktiviert.IsEnabled = True
End Sub

Private Sub btnDeaktivieren_Click(sender As Object, e As RoutedEventArgs)
    btnBeimAnzeigenDeaktiviert.IsEnabled = False
End Sub
```

Das Ergebnis sieht dann wie in Bild 1 aus.

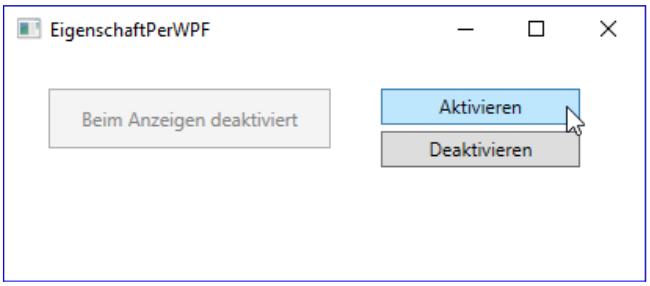


Bild 1: Aktivieren und deaktivieren per Schaltfläche und VB-Prozedur

Die nächste Möglichkeit ist, diese Eigenschaft an eine öffentliche Eigenschaft im Code behind-Modul zu koppeln.

Eigenschaft an Code behind-Eigenschaft koppeln

Sie können die Eigenschaft **IsEnabled** auch an eine öffentlich deklarierte Eigenschaft im Code behind-Modul koppeln (siehe

Fenster **EigenschaftPerCodeBehind.xaml**). Man könnte naiverweise meinen, dass folgender Code im Code behind-Modul ausreicht:

```
Public Property Aktiviert As Boolean

Private Sub btnAktivieren_Click(sender As Object, e As RoutedEventArgs)
    Aktiviert = True
End Sub

Private Sub btnDeaktivieren_Click(sender As Object, e As RoutedEventArgs)
    Aktiviert = False
End Sub
```

Dabei wollen wir voraussetzen, dass die Schaltfläche über die Eigenschaft **IsEnabled** an die Eigenschaft **Aktiviert** gebunden ist:

```
<Button x:Name="btnBeimAnzeigenDeaktiviert" Content="Beim Anzeigen deaktiviert" IsEnabled="{Binding Aktiviert}" ... />
```

Das reicht allerdings nicht aus, denn beim Anklicken der beiden Schaltflächen **btnAktivieren** und **btnDeaktivieren** geschieht nichts. Das ändert sich auch nicht, wenn wir die Schnittstelle **INotifyPropertyChanged** implementieren, die notwendig ist, um Aktualisierungen von Eigenschaften über die Bindung in die Benutzeroberfläche zu übertragen. Zusätzlich benötigen wir nämlich auch noch eine Konstruktor-Methode, die beim Erstellen des Fensters aufgerufen wird und dem WPF-Fenster mitteilt, dass es an die Eigenschaften des Code behind-Moduls gebunden werden soll. Insgesamt sieht der Code für das Code behind-Modul nun wie folgt aus. Als Erstes fügen wir einen Verweis auf den Namespace **System.ComponentModel** hinzu, den wir für die **INotifyPropertyChanged**-Schnittstelle benötigen:

```
Imports System.ComponentModel
```

Dann fügen wir die **Implements**-Anweisung zur Implementierung der Schnittstelle hinzu sowie die Eigenschaft **Aktiviert**, in der wir die Schnittstelle nutzen wollen:

```
Public Class EigenschaftPerCodeBehind
    Implements INotifyPropertyChanged
    Private _Aktiviert As Boolean
    Public Event PropertyChanged As PropertyChangedEventHandler Implements INotifyPropertyChanged.PropertyChanged

    Public Property Aktiviert As Boolean
        Get
            Return _Aktiviert
        End Get
        Set
```

```
            _Aktiviert = Value
            RaiseEvent PropertyChanged(Me, New PropertyChangedEventArgs("Aktiviert"))
        End Set
    End Property
```

Fehlt noch die Konstruktor-Anweisung, in der wir mit **DataContext = Me** angeben, dass die öffentlichen Eigenschaften des Code behind-Moduls vom XAML-Code aus gebunden werden können:

```
    Public Sub New()
        InitializeComponent()
        DataContext = Me
        Aktiviert = False
    End Sub
End Class
```

Das allein reicht allerdings noch nicht aus. Wir müssen auch noch die Eigenschaft **IsEnabled="{Binding Aktiviert}"** ersetzen, denn für die Bindung an eine Eigenschaft einer anderen Klasse ist ein DataTrigger nötig. Diesen hinterlegen wir wie folgt für die Schaltfläche **btnAktivierenDeaktivieren**:

```
<Button x:Name="btnAktivierenDeaktivieren" Content="Aktivieren/Deaktivieren" HorizontalAlignment="Left" Height="38"
        Margin="28,22,0,0" VerticalAlignment="Top" Width="180">
    <Button.Style>
        <Style TargetType="{x:Type Button}">
            <Setter Property="IsEnabled" Value="False"></Setter>
            <Style.Triggers>
                <DataTrigger Binding="{Binding Aktiviert}" Value="True">
                    <Setter Property="IsEnabled" Value="True" />
                </DataTrigger>
            </Style.Triggers>
        </Style>
    </Button.Style>
</Button>
```

Hier sind gleich mehrere wichtige Informationen untergebracht:

- Mit dem ersten **Setter**-Element legen wir für die Eigenschaft **IsEnabled** den Wert **False** als initialen Wert fest. Das ist die Methode, den Standardwert zu definieren, wenn Sie einen DataTrigger verwenden.

- Im **DataTrigger**-Element definieren wir, welchen Wert wir untersuchen wollen, hier die gebundene Eigenschaft **Aktiviert** für das Attribut **Property**, und mit welchem Wert wir den gelieferten Wert vergleichen wollen – hier **True** für das Attribut **Value**.

Ist diese Bedingung erfüllt, also ist der Wert der Eigenschaft **Aktiviert** gleich **True**, wird die im untergeordneten Setter-Element angegebene Eigenschaft mit dem dort angegebenen Wert gefüllt.

Achtung: Wenn Sie den Standardwert bei Verwendung eines DataTriggers mit WPF-Mitteln einstellen wollen, gelingt dies nur mit dem erwähnten **Setter**-Element – das Einstellen der Eigenschaft als Attribut des hier verwendeten **Button**-Elements ist nicht möglich (also etwa mit **IsEnabled = False**). Wenn Sie das dennoch tun, kann der Wert durch den DataTrigger nicht mehr geändert werden!

Mehrere Bedingungen verwenden

Wie gehen wir nun vor, wenn wir mehrere Bedingungen verwenden wollen? Wir starten mit zwei Oder-verknüpften Bedingungen. Dazu fügen wir zu einem neuen Fenster (siehe **EigenschaftMitOder.xaml**) die folgenden beiden Kontrollkästchen hinzu:

```
<CheckBox x:Name="chkBedingung1" Content="Bedingung
1" HorizontalAlignment="Left" Margin="239,23,0,0"
VerticalAlignment="Top"/>
<CheckBox x:Name="chkBedingung2" Content="Bedingung
2" HorizontalAlignment="Left" Margin="239,43,0,0"
VerticalAlignment="Top"/>
```

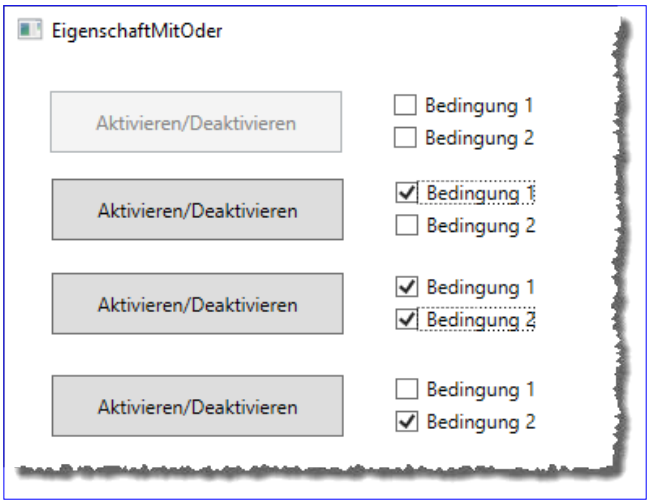


Bild 2: Bedingung für das Aktivieren mit Oder

Diese sollen nun Oder-verknüpft dafür sorgen, dass bei mindestems einem aktivierten Kontrollkästchen die Schaltfläche **btnAktivierenDeaktivieren** aktiviert wird – so wie in Bild 2.

Statt eines **DataTrigger**-Elements verwenden wir nun gleich zwei – einen für jede Bedingung. Wir stellen den Standardwert für die Eigenschaft **IsEnabled** wieder auf **False** ein. Die beiden **DataTrigger**-Elemente prüfen jeweils den Wert der Eigenschaft **IsChecked** eines der Kontrollkästchen und stellen dann den Wert des Attributes **IsEnabled** auf **True** ein. Dadurch, dass die **DataTrigger** nacheinander durchlaufen werden, reicht es aus, wenn einer der **DataTrigger** ausgelöst wird. Somit arbeiten die Trigger wie eine Oder-Verknüpfung:

```
<Button x:Name="btnAktivierenDeaktivieren" Content="Aktivieren/Deaktivieren" ...>
    <Button.Style>
        <Style TargetType="{x:Type Button}">
            <Setter Property="IsEnabled" Value="False"/>
            <Style.Triggers IsEnabled="Test">
                <DataTrigger Binding="{Binding ElementName=chkBedingung1, Path=IsChecked}" Value="True">
                    <Setter Property="IsEnabled" Value="True" />
                </DataTrigger>
                <DataTrigger Binding="{Binding ElementName=chkBedingung2, Path=IsChecked}" Value="True">
                    <Setter Property="IsEnabled" Value="True" />
```

Mehrspaltige Kombinationsfelder

Manchmal möchte man mit einem Kombinationsfeld nicht nur den Wert eines Feldes anzeigen, sondern gleich mehrere. Unter Access war das zumindest für die Auswahlliste schnell erledigt – Sie brauchten einfach nur die Eigenschaften Spaltenanzahl und Spaltenbreiten anzupassen. Wenn der Inhalt des Kombinationsfeldes auch die Daten mehrerer Felder liefern sollte, mussten Sie diese per Zeichenkettenfunktion zusammenführen. Wie aber sieht das im WPF-Steuerelement »ComboBox« aus? Dieser Artikel zeigt, welche Möglichkeiten dieses Steuerelement für mehrspaltige Einträge bietet.

Vorbereitung

Für das Beispiel verwenden wir die Basis vieler anderer Artikel: Wir verwenden die aus einer Access-Datenbank erzeugten Entitätsklassen plus den von dort exportierten Daten. Sie brauchen, wenn Sie das Beispielprojekt verwenden wollen, einfach nur die Paket-Manager-Konsole anzuzeigen und den folgenden Befehl einzugeben:

```
Update-Database
```

Das fügt dann eine neue Datenbank entsprechend der Verbindungszeichenfolge aus der Datei **App.config**, die Sie gegebenenfalls noch anpassen müssen, zum SQL Server beziehungsweise zu **LocalDb** hinzu.

Im Code behind-Modul fügen wir folgenden Code hinzu:

```
Imports System.Collections.ObjectModel0

Class MainWindow
    Private m_Kunden As List(Of Kunde)
    Private dbContext As BestellverwaltungContext
    Public Sub New()
        InitializeComponent()
        dbContext = New BestellverwaltungContext
        Kunden = New List(Of Kunde)(dbContext.Kunden)
        DataContext = Me
    End Sub

    Public Property Kunden As List(Of Kunde)
        Get
            Return m_Kunden
        End Get
        Set(value As List(Of Kunde))
            m_Kunden = value
```

```
        End Set
    End Property
End Class
```

Danach fügen wir das folgende **ComboBox**-Element zum Startfenster **MainWindow.xaml** hinzu (siehe **cboKundeID_EinFeld**):

```
<ComboBox x:Name="cboKundeID_EinFeld" Padding="0" Height="21"
Margin="118,111,0,0" Width="78"
    ItemsSource = "{Binding Kunden}">
    <ComboBox.ItemTemplate>
        <DataTemplate>
            <TextBlock>
                <TextBlock.Text>
                    <MultiBinding StringFormat="{}{0}">
                        <Binding Path="Nachname" />
                    </MultiBinding>
                </TextBlock.Text>
            </TextBlock>
        </DataTemplate>
    </ComboBox.ItemTemplate>
</ComboBox>
```

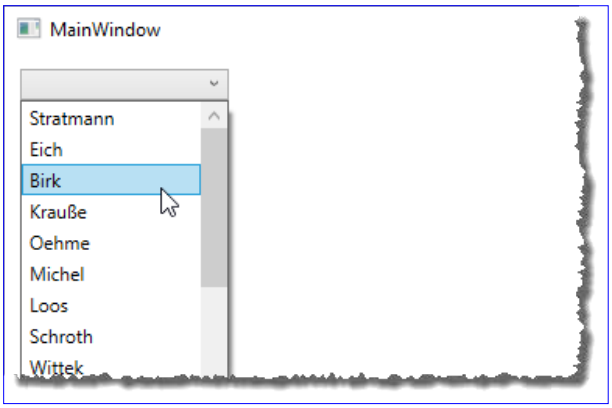


Bild 1: Kombinationsfeld mit einer Spalte

Damit erhalten wir das Kombinationsfeld aus Bild 1. Um den Inhalt eines zweiten Feldes als ausgewählten Eintrag anzuzeigen wie etwa **Vorname**, passen Sie das **MultiBinding**-Element einfach wie folgt an (siehe):

```
<MultiBinding StringFormat="{}{0},{1}">
    <Binding Path="Nachname" />
    <Binding Path="Vorname" />
</MultiBinding>
```

Das Ergebnis sieht dann wie in Bild 2 aus. Sowohl der ausgewählte Eintrag als auch die Einträge der Liste werden dann im gewünschten Format **<Nachname>,<Vorname>** angezeigt.

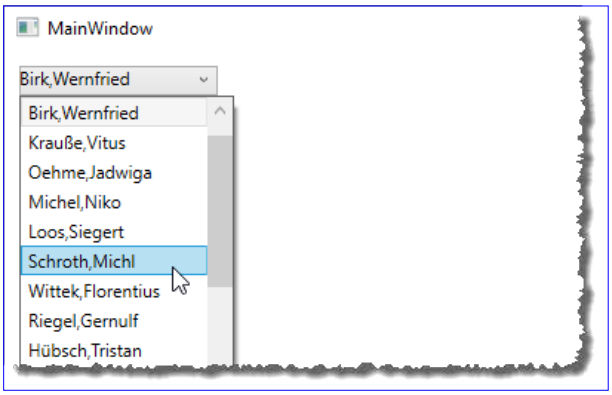


Bild 2: Kombinationsfeld mit einer Spalte und zwei kombinierten Feldern

Mehrspaltige Anzeige

Um die Anzeige nun auch noch auf mehrere Spalten aufzuteilen, verwenden Sie ein etwas aufwendigeres Konstrukt. Dabei weisen wir dem Unterelement **ComboBox.ItemTemplate** das **DataTemplate**-Element zu, das ein **StackPanel** mit den drei **TextBlock**-Elementen für unsere drei Spalten aufnimmt (siehe **cboKunden_MehrereSpaltenNichtBuendig**):

```
<ComboBox x:Name="cboKunden_MehrereSpaltenNichtBuendig" ItemsSource="{Binding Kunden}" Margin="157,10,311,381" >
    <ComboBox.ItemTemplate>
        <DataTemplate>
            <StackPanel Orientation="Horizontal">
                <TextBlock Margin="5" Grid.Column="0" Text="{Binding Vorname}"></TextBlock>
                <TextBlock Margin="5" Grid.Column="1" Text="{Binding Nachname}"></TextBlock>
                <TextBlock Margin="5" Grid.Column="2" Text="{Binding EMail}"></TextBlock>
            </StackPanel>
        </DataTemplate>
    </ComboBox.ItemTemplate>
</ComboBox>
```

Wenn wir das Beispielprojekt nun starten, zeigt die Liste bereits die gewünschten Daten an (siehe Bild 3).

Allerdings bleibt das Kombinationsfeld nach der Auswahl eines der Einträge leer. Außerdem werden die Elemente der drei Spalten nicht bündig angezeigt.

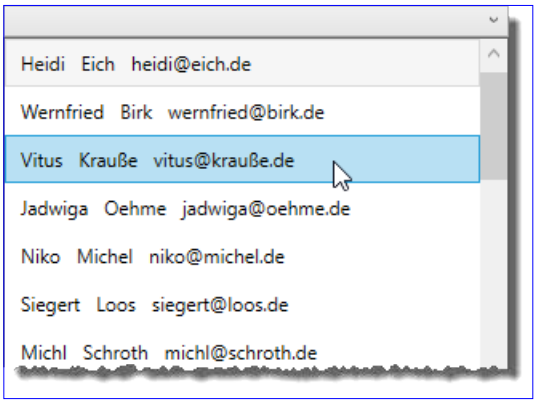


Bild 3: Kombinationsfeld mit drei Spalten, die allerdings nicht bündig sind

ComboBox-Spalten per Grid

Also versuchen wir eine Alternative. Diese verwendet innerhalb des **DataTemplate**-Elements ein **Grid** mit den **ColumnDefinition**-Elementen für die drei Spalten, in denen wir feste Spaltenbreiten definieren (siehe **cboKunden_MehrereSpaltenFixeBreite**):

```
<ComboBox x:Name="cboKunden_MehrereSpaltenFixeBreite" ItemsSource="{Binding Kunden}" Margin="157,10,311,381">
    <ComboBox.ItemTemplate>
        <DataTemplate>
            <Grid>
                <Grid.ColumnDefinitions>
                    <ColumnDefinition Width="100"></ColumnDefinition>
                    <ColumnDefinition Width="100"></ColumnDefinition>
                    <ColumnDefinition Width="150"></ColumnDefinition>
                </Grid.ColumnDefinitions>
                <TextBlock Margin="5" Grid.Column="0" Text="{Binding Vorname}"></TextBlock>
                <TextBlock Margin="5" Grid.Column="1" Text="{Binding Nachname}"></TextBlock>
                <TextBlock Margin="5" Grid.Column="2" Text="{Binding EMail}"></TextBlock>
            </Grid>
        </DataTemplate>
    </ComboBox.ItemTemplate>
</ComboBox>
```

Entity Framework: Bilder in WPF

Im Artikel »PowerApps: Bilder in Datenbank speichern« haben wir gezeigt, wie Sie mit einer PowerApp über ein Smartphone oder ein Tablet Bilder aufnehmen und diese dann in einer Datenbank speichern. Nun wollen wir uns ansehen, wie Sie diese Bilder aus der Datenbank in einer WPF-Anwendung anzeigen, die über das Entity Framework auf die Tabellen der Datenbank zugreift.

Voraussetzung

Um die Beispiele dieses Artikels nachzuvollziehen, benötigen Sie eine Datenbank, die eine Tabelle namens **Fotos** enthält – genau wie die, die wir im oben genannten Artikel erstellt haben. Diese enthält ein Primärschlüsselfeld namens **ID** und mit dem Datentyp **int** sowie ein Feld namens **Foto** und mit dem Datentyp **image**. Unsere Beispieldatenbank liegt auf einem Azure-Server. Sie können aber natürlich auch unsere Beispieldatenbank mit ein paar Beispielfotos aus dem SQL-Skript aus dem Download auf einem lokalen SQL Server anlegen und mit der folgenden Anwendung auf diese Datenbank zugreifen.

Vorbereitung

Wir erstellen in Visual Studio ein neues Projekt auf Basis der Vorlage **Visual Basic\Windows Desktop\WPF-App**. Da wir diesmal nicht, wie so oft in den vorherigen Ausgaben, ein Code First-Model erstellen, sondern mit einer bestehenden Datenbank starten, noch einmal eine kurze Beschreibung der nötigen Schritte. Dem neuen Projekt fügen wir als Erstes ein neues Element hinzu und wählen den Typ **ADO.NET Entity Data Model** aus. Nachdem wir als Name **FotoverwaltungContext** eingestellt haben, klicken wir auf **Hinzufügen** (siehe Bild 1).

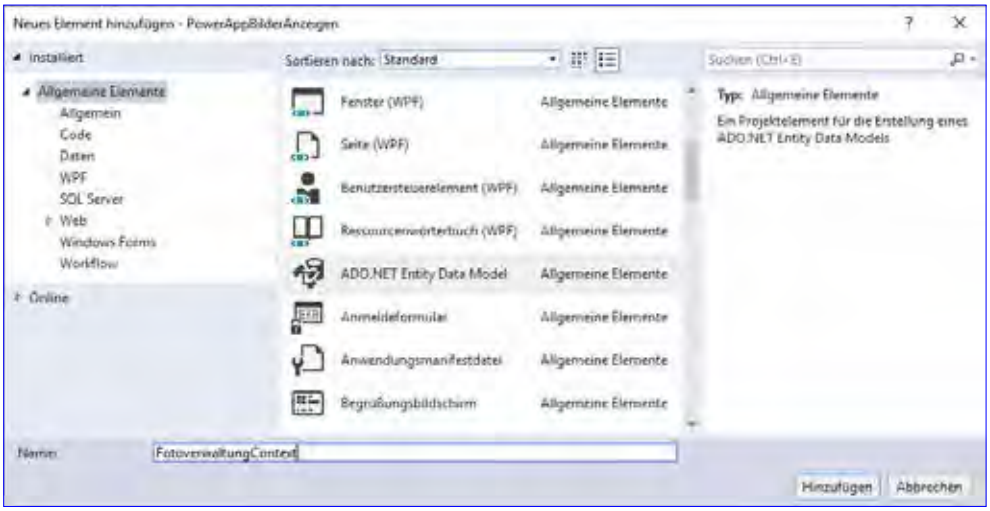


Bild 1: Einfügen des Entity Data Models

Danach wählen wir als Modellinhalt **Code First aus**

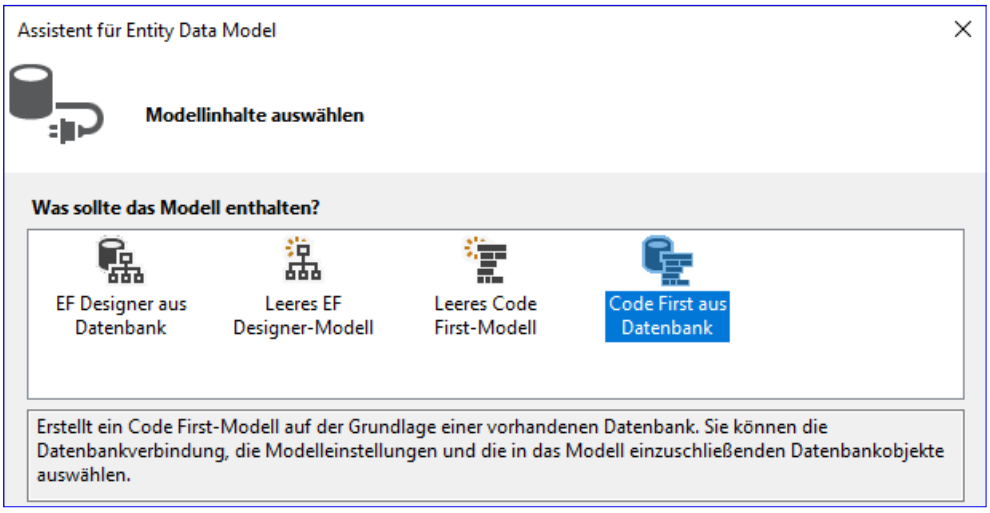


Bild 2: Auswahl des Typs **Code First aus Datenbank**

Datenbank aus (siehe Bild 2). Der folgende Dialog bietet die Möglichkeit zur Auswahl der Datenverbindung. Hier klicken wir auf die Schaltfläche **Neue Verbindung**. Im nun erscheinenden Dialog namens **Verbindungseigenschaften** geben Sie unter **Servername** den Namen des gewünschten Servers ein. Danach wählen Sie die Authentifizierungsmethode aus – beim Zugriff auf eine Azure-Datenbank immer die SQL Server-Authentifizierung – und geben die Zugangsdaten ein.

Schließlich wählen Sie die Datenbank aus und bestätigen nach einem erfolgreichen Test mit einem Klick auf die Schaltfläche **Testverbindung** die Eingaben mit **OK** (siehe Bild 3).

Danach kehren wir zum Dialog Assistent für Entity Data Model zurück, der Sie fragt, ob Sie Benutzername und Kennwort wirklich in der Verbindungszeichenfolge speichern wollen (siehe Bild 4). Für dieses Beispiel lassen wir diese Daten in der Verbindungszeichenfolge, aber im Artikel **Zugangsdaten für SQL Server abfragen** zeigen wir, wie Sie diese jeweils beim Start der Anwendung abfragen können.

Schließlich wählen wir im Schritt **Wählen Sie Ihre Datenbankobjekte und Einstellungen** noch die Tabelle aus, deren Daten wir in der Anwendung anzeigen wollen. Wir haben die Tabelle **tblFotos** einfach in eine bestehende Datenbank eingefügt, daher müssen wir diese einzeln auswählen (siehe Bild 5).

Dieser Dialog bietet auch noch die Option **Generierte Objektnamen in den Singular oder Plural setzen** an. Da die Tabelle das Präfix **tbl** im Namen **tblFotos** enthält, das wir in unserem Entity Data Model nicht als Bezeichnung der Entität sehen wollen, müssen wir hier ohnehin noch ein Mapping vornehmen.

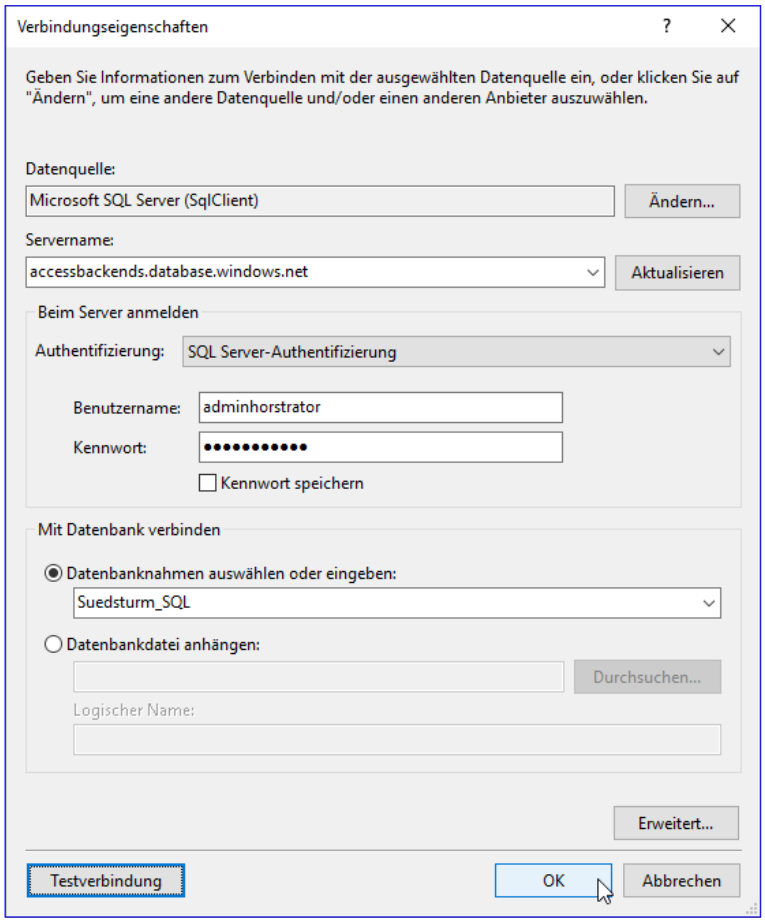


Bild 3: Auswahl der Datenquelle

Bild 4: Einstellen weiterer Eigenschaften

Nach einem Klick auf die Schaltfläche **Fertigstellen** finden Sie zwei neue Elemente im Projektmappen-Explorer vor: die Klasse **FotoverwaltungContext.vb**, die von der Klasse **DbContext** erbt und unter anderem das **DbSet** namens **tblFotos** für den Zugriff auf die Liste der Entitäten auf Basis der in der Tabelle gespeicherten Daten bereitstellt sowie die Klasse **tblFotos.vb** als Entitätsklasse für einen Datensatz der Tabelle **tblFotos**.

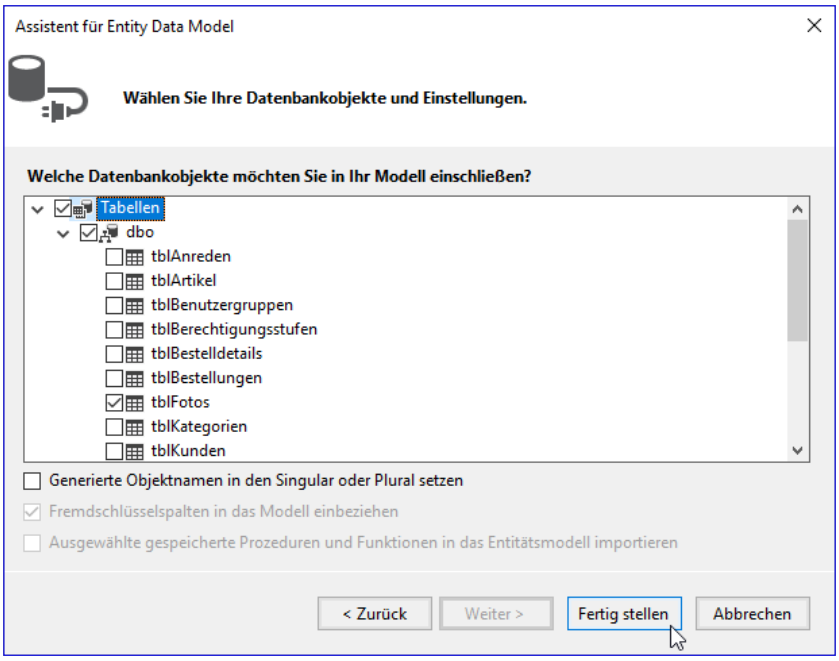


Bild 5: Auswahl der gewünschten Tabelle

Die Benennung der Klasse und des Namens des **DbSet**-Elements entspricht nicht unseren Vorstellungen, also ändern wir das ein wenig ab. Wie das genau gelingt, zeigen wir Ihnen im Artikel **EDM für bestehende Datenbank mit Code First**. Zum schnellen Nachbauen hier die notwendigen Änderungen. Als Erstes ändern wir den Namen der Klasse **tblFotos.vb** in **Foto.vb**.

In der Klasse **Foto.vb** müssen wir den Namen der Eigenschaft **Foto** in Fotodaten ändern, da keine Eigenschaft genauso heißen darf wie die Klasse:

```
Partial Public Class Foto
    Public Property ID As Integer
    <Column(TypeName:="image")>
    <Required>
    Public Property Fotodaten As Byte()
End Class
```

Damit die Klasse **Foto** auf die Tabelle **tblFotos** gemappt wird und die Eigenschaft **Fotodaten** der Klasse auf das Feld **Foto** der Tabelle, fügen wir folgende Zeilen zur Methode **OnModelCreating** der Klasse **FotoverwaltungContext.vb** hinzu:

```
Protected Overrides Sub OnModelCreating(ByVal modelBuilder As DbModelBuilder)
    modelBuilder.Entity(Of Foto)().
        ToTable("tblFotos").
        Property(Function(t) t.Fotodaten).HasColumnName("Foto")
End Sub
```

Damit können wir nun testen, ob der Zugriff auf die Datensätze der Tabelle **tblFotos** gelingt. Dazu fügen wir der Klasse **MainWindow.xaml.vb** zunächst eine Konstruktor-Methode hinzu. In dieser erstellen wir einen Datenbankkontext für den Zugriff auf

die Datenbank. Außerdem versuchen wir, auf das erste Element der Auflistung **Fotos** dieses Kontextes zuzugreifen und den Inhalt der Eigenschaften **ID** und **Fotodaten** in einem Meldungsfenster auszugeben:

```
Class MainWindow
    Public Sub New()
        Dim dbContext As FotoverwaltungContext
        dbContext = New FotoverwaltungContext
        Dim foto As Foto
        foto = dbContext.Fotos.First()
        MessageBox.Show(foto.ID.ToString() + vbCrLf + vbCrLf + foto.Fotodaten.ToString())
    End Sub
End Class
```

Sofern Sie bereits einen Datensatz zu dieser Tabelle hinzugefügt haben, werden die entsprechenden Informationen beim Start der Anwendung ausgegeben.

Bild im WPF-Fenster anzeigen

Nun schauen wir uns an, wie wir den Inhalt der Eigenschaft **Fotodaten**, also des Feldes **Foto** der Tabelle **tblFotos**, in einem Steuerelement in einem WPF-Fenster anzeigen. Dazu benötigen wir zunächst ein geeignetes Steuerelement. Wir fügen dem Fenster **MainWindow.xaml** zunächst ein **Image**-Steuerelement hinzu, das wir mit dem Bild aus der Eigenschaft **Fotodaten** füllen wollen. Das Steuerelement ziehen wir einfach aus der Toolbox in das WPF-Fenster. Dadurch entsteht etwa der folgende XAML-Code:

```
<Image HorizontalAlignment="Left" Height="288" Margin="112,50,0,0" VerticalAlignment="Top" Width="364"/>
```

Das Bild ist im Feld **Foto** der Tabelle **tblFotos** unter dem Datentyp **image** gespeichert (siehe Bild 6). Wir müssen nun einen Weg finden, den Inhalt dieses Feldes im **Image**-Steuerelement anzuzeigen. Dazu verarbeiten wir dieses zunächst in der Code behind-Klasse **MainWindow.xaml.vb**. Hier verschieben wir die Deklaration des Datenbankkontextes zunächst aus der Konstruktor-Methode in den allgemeinen Teil der Klasse:

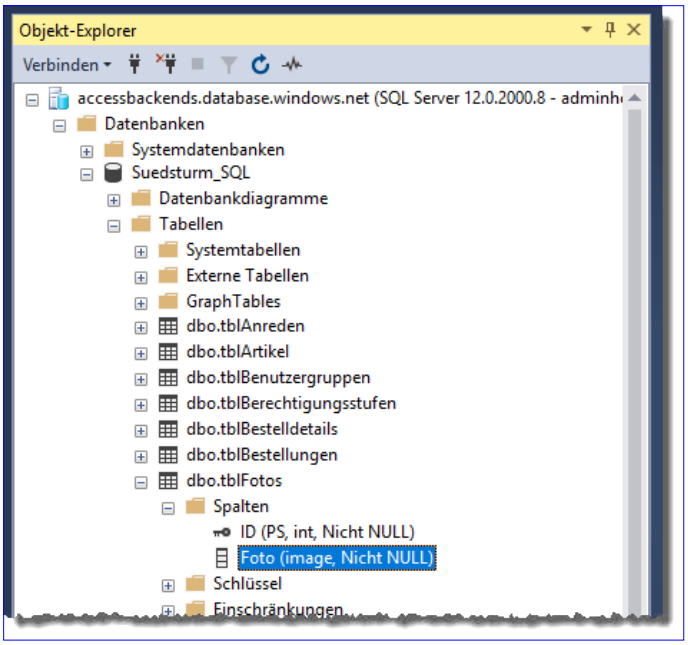


Bild 6: Das Feld **Foto** in der Tabelle **tblFotos**

```
Class MainWindow
    Dim dbContext As FotoverwaltungContext
```

Dann deklarieren wir eine private Variable namens **m_Fotodaten** mit dem Datentyp **BitmapImage**, die wir über die folgende **Property** namens **Fotodaten** mit dem gleichen Datentyp für den lesenden und schreibenden Zugriff verfügbar machen:

```
Dim m_Fotodaten As BitmapImage
Public Property Fotodaten As BitmapImage
    Get
        Return m_Fotodaten
    End Get
    Set(value As BitmapImage)
        m_Fotodaten = value
    End Set
End Property
```

Danach erweitern wir die Konstruktor-Methode **New**. Nach dem Zuweisen des Datenbankkontextes zur Variablen dbContext deklarieren wir ein Objekt des Typs **Foto** und weisen diesem das erste Element der mit der Auflistung **Fotos** eingelesenen Tabelle **tblFotos** zu:

```
Public Sub New()
    dbContext = New FotoverwaltungContext
    Dim foto As Foto
    foto = dbContext.Fotos.First()
```

Dann lesen wir zunächst den Inhalt der Eigenschaft **Fotodaten** (also eigentlich des Feldes **Foto** der Tabelle **tblFotos**) in ein Objekt des Typs **Byte** mit dem Namen **blob** ein. Danach erstellen wir ein Objekt namens **stream** mit dem Typ **MemoryStream**, in das wir den Inhalt des Byte-Arrays **blob** mit der **Write**-Methode schreiben.

Dann stellen wir die Position des **Stream**-Objekts auf die Position **0** ein:

```
    Dim blob As Byte() = foto.Fotodaten
    Dim stream As MemoryStream = New MemoryStream()
    stream.Write(blob, 0, blob.Length)
    stream.Position = 0
```

Nun kommt erstmals das Objekt des Typs **BitmapImage** ins Spiel, das wir anschließend auch dem **Image**-Steuerelement als Quelle zuweisen können. Dieses erstellen wir unter dem Namen **img** und initialisieren es mit der Methode **BeginInit**.

Dann weisen wir als **StreamSource** den Inhalt der **MemoryStream**-Variablen **stream** zu und beenden die Initialisierung von **img** mit der **EndInit**-Methode:

```
    Dim img As BitmapImage
    img = New BitmapImage
    img.BeginInit()
    img.StreamSource = stream
    img.EndInit()
```

Schließlich weisen wir **img** der öffentlichen Eigenschaft **Fotodaten** zu und stellen die aktuelle Klasse als **DataContext** der **WPF**-Seite ein:

```
        Fotodaten = img
        DataContext = Me
    End Sub
End Class
```

Danach müssen wir nur noch der Eigenschaft **Source** des **Image**-Elements im XAML-Code den Pfad zur Eigenschaft **Fotodaten** mitteilen:

```
<Image HorizontalAlignment="Left" Height="288" Margin="112,50,0,0"
VerticalAlignment="Top" Width="364" Source="{Binding Path=Fotodaten}"/>
```

Das Ergebnis sieht dann wie in Bild 7 aus.

Bild 7: Ein Bild im WPF-Fenster

Durch Bilder blättern

Wenn wir nun nicht nur ein Bild anzeigen, sondern auch durch alle vorhandenen Bilder blättern wollen, benötigen wir prinzipiell die gleichen Techniken, die wir auch sonst zum Blättern in Datensätzen verwenden. Dazu erstellen wir ein neues Fenster namens **FotosBlaettern.xaml**. Dieses öffnen wir vom Startfenster **MainWindow.xaml** aus mit einer neuen Schaltfläche, der wir die folgenden Codezeilen hinzufügen:

```
Private Sub btnBeispielZumBlaettern_Click(sender As Object, e As RoutedEventArgs)
    Dim wnd As FotosBlaettern
    wnd = New FotosBlaettern
    wnd.ShowDialog()
End Sub
```

Die Techniken, die wir weiter oben kennengelernt haben, können wir für das Blättern durch Datensätze mit Bildern ebenfalls verwenden. Außerdem nutzen wir einige Techniken, die wir im Artikel **EDM: Blättern in Datensätzen** vorgestellt haben. Im XAML-Code verwenden wir in den Ressourcen einige DataTrigger, mit denen wir die Schaltflächen aktivieren und deaktivieren wollen (siehe ebenfalls im oben genannten Artikel):

```
<Window x:Class="FotosBlaettern" ... Title="FotosBlaettern" Height="450" Width="800">
    <Window.Resources>
        <Style x:Key="ButtonVorheriger" TargetType="{x:Type Button}">
            <Style.Triggers>
                <DataTrigger Binding="{Binding Erster}" Value="False">
                    <Setter Property="IsEnabled" Value="False"></Setter>
                </DataTrigger>
            </Style.Triggers>
```

EDM für bestehende Datenbank mit Code First

Wenn Sie ein Entity Data Model mit der Vorlage »Code First aus Datenbank« auf Basis einer bestehenden Datenbank erstellen, haben Sie vielleicht Pech und die Namen der Tabellen der Datenbank und der enthaltenen Felder lauten nicht so, wie Sie die Entitätsklassen, die DbSet-Elemente und die Eigenschaften der Klassen nennen möchten. Dann haben Sie verschiedene Möglichkeiten: Zum Beispiel können Sie die Bezeichnungen in der Datenbank anpassen. Das geht aber oft nicht, weil vielleicht noch andere Frontends auf die gleiche Datenbank zugreifen. Dann haben Sie noch die Möglichkeit, die Bezeichnungen von Datenbank und Entity Data Model so zu mappen, dass beide Seiten zufrieden sind. Wie letzteres gelingt, zeigen wir im vorliegenden Artikel am Beispiel der Südsturm-Datenbank.

Einfaches Beispiel: tblFotos

Wir starten mit einem sehr einfachen Beispiel, nämlich der Tabelle **tblFotos**. Diese haben wir als Tabelle zum Speichern von Fotos erstellt, die wir mit einer PowerApp über das Smartphone aufnehmen wollen. Dabei haben wir die Tabelle leichtsinnigerweise **tblFotos** genannt, statt einfach **Fotos** ohne Präfix. Wenn wir nun ein Entity Data Model erstellen, erhalten wir für die Klasse **FotoverwaltungContext.db** etwa den folgenden Code (siehe Projekt **BilderWPF**, Beispieldatenbank in **Suedsturm_SQL.sql**):

```
Imports System.ComponentModel.DataAnnotations.Schema

Partial Public Class FotoverwaltungContext
    Inherits DbContext
    Public Sub New()
        MyBase.New("name=FotoverwaltungContext")
    End Sub
    Public Overridable Property tblFotos As DbSet(Of tblFotos)
    Protected Overrides Sub OnModelCreating(ByVal modelBuilder As DbModelBuilder)
    End Sub
End Class
```

Hier hätten wir gern den Namen der Property für das **DbSet** so geändert, dass es **Fotos** statt **tblFotos** heißt und Elemente des Typs **Foto** enthält (die auch noch **tblFotos** heißen). In der Entitätsklasse **tblFotos** geht es so weiter. Diese hat nach dem Erstellen des Entity Data Models den folgenden Code erhalten:

```
Imports System.ComponentModel.DataAnnotations
Imports System.ComponentModel.DataAnnotations.Schema

Partial Public Class tblFotos
    Public Property ID As Integer
```

```
    <Column(TypeName:="image")>
    <Required>
    Public Property Foto As Byte()
End Class
```

Wir wollen dies Schritt für Schritt so anpassen, dass wir mit einer Konstruktor-Methode für das Fenster **MainWindow** wie der folgenden auf die Daten zugreifen können:

```
Class MainWindow
    Public Sub New()
        Dim dbContext As FotoverwaltungContext
        dbContext = New FotoverwaltungContext
        Dim foto As Foto
        foto = dbContext.Fotos.First()
        MessageBox.Show(foto.ID.ToString())
    End Sub
End Class
```

Wir wollen also ein **DbSet** namens **Fotos** verwenden und damit auf Elemente des Typs **Foto** zugreifen. Dazu müssen wir dem Entity Data Model auf irgendeine Weise mitteilen, dass es das **DbSet** namens **Fotos** auf die Tabelle **tblFotos** mappen soll und die Klasse **Foto** auf die einzelnen Datensätze.

Mapping in der Methode OnModelCreating

Der richtige Ort für ein solches Mapping ist die Methode **OnModelCreating**, die beim Erstellen des Entity Data Models auf Basis der Vorlage **Code First aus Datenbank** automatisch in der Klasse **FotoverwaltungContext** angelegt wurde. Der erste Schritt ist das Umbenennen der Klasse **tblFotos** in **Foto**. Das erledigen wir ganz einfach, indem wir den entsprechenden Eintrag im Projektmappen-Explorer markieren, diesen nochmals anklicken und dann die Bezeichnung ändern. Danach erscheint noch eine Meldung, die fragt, ob Verweise auf das Codeelement angepasst werden sollen (siehe Bild 1).

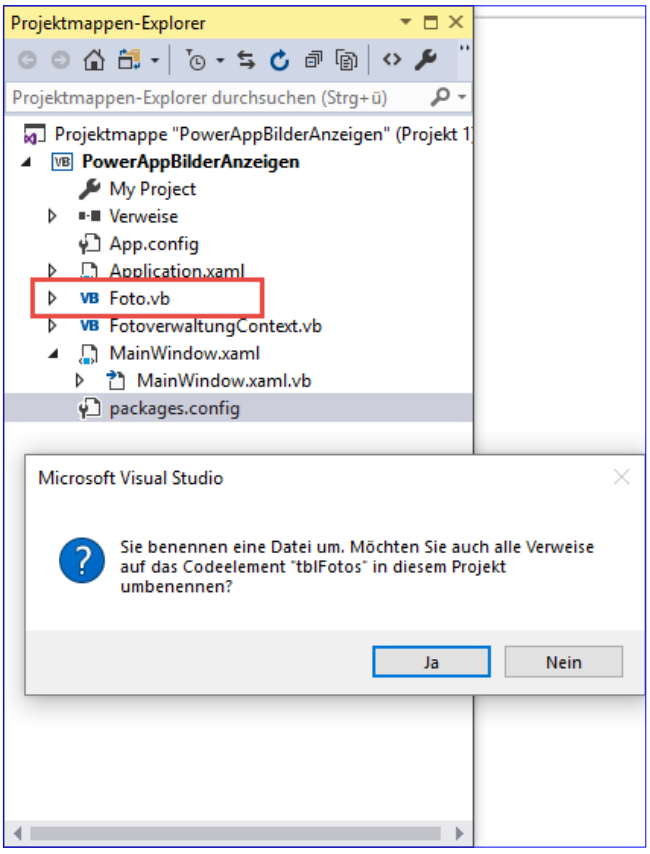


Bild 1: Ändern eines Klassennamens

Wenn Sie hier auf **Ja** klicken, werden in unserer kleinen Beispielanwendung folgende Änderungen durchgeführt:

- Die Bezeichnung der Klasse wird ebenfalls in **Foto** geändert.

- In der Klasse **FotoverwaltungContext** wird der Typ der Klasse des **DbSets** ebenfalls geändert:

```
Partial Public Class FotoverwaltungContext
    Inherits DbContext
    ...
    Public Overridable Property tblFotos As DbSet(Of Foto)
    ...
End Class
```

In dieser Klasse sind dann weitere Änderungen nötig. Anschließend sieht die oben bereits teilweise geänderte Zeile mit der Definition des **DbSet** wie folgt aus:

```
Public Overridable Property Fotos As DbSet(Of Foto)
```

Mapping hinzufügen

Damit passen die Deklarationen der Klasse und des **DbSet**-Elements nun zu dem Code, den wir für die Konstruktor-Methode unseres Fensters **MainWindow.xaml** erstellt haben. Was geschieht nun, wenn wir die Anwendung starten?

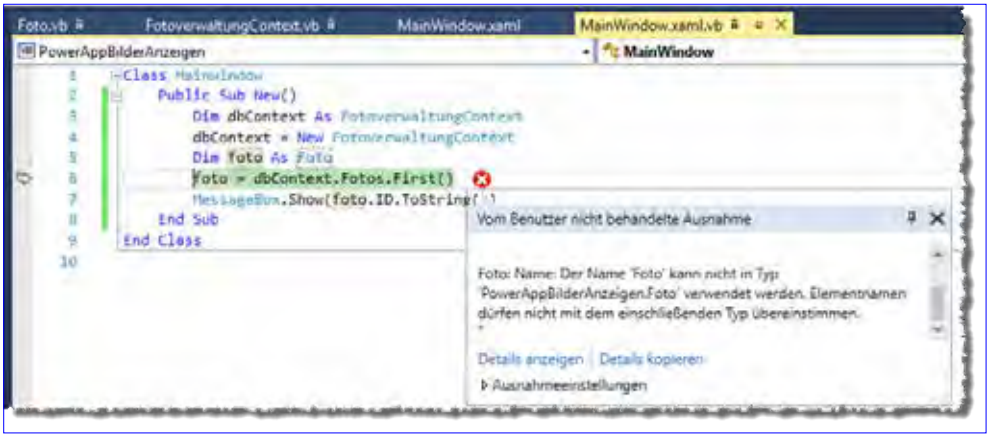


Bild 2: Fehler beim Zugriff auf die Klasse

Wir erhalten einen unerwarteten Fehler: Visual Studio bemängelt, dass wir eine Eigenschaft namens **Foto** in der gleichnamigen Klasse verwenden (siehe Bild 2). Damit erhalten wir also noch ein Problem, das aus der Benennung der Tabellen und Felder der Beispieldatenbank resultiert. Das Feld **Foto** können wir nicht mit dem Eigenschaftsnamen **Foto** ansprechen, da eine Klasse keine Eigenschaften besitzen darf, die genauso heißen wir die Klasse selbst. Also ändern wir den Namen der Eigenschaft in der Klasse **Foto** auf **Fotodaten**:

```
Partial Public Class Foto
    ...
    Public Property Fotodaten As Byte()
End Class
```

Nach einem erneuten Start der Anwendung erhalten wir dann die Fehler, mit denen wir gerechnet hätten. Der erste lautet wie folgt und er tritt beim Zugriff auf die Daten über **dbContext.Fotos.First** auf (siehe auch Bild 3):

```
System.InvalidOperationException: "Die Sequenz enthält keine Elemente."
```

Das ist etwas überraschend, denn wir hatten mit einem Fehler gerechnet, der durch eine fehlende Tabelle ausgelöst wird. Schauen wir uns die Eigenschaften der Auflistung **Fotos** wie in Bild 4 an, sehen wir, dass das Entity Framework anscheinend versucht, auf eine Tabelle namens **Fotoes** zuzugreifen. Das ist offensichtlich die Plural-Form der Klasse **Foto**, die Entity Framework automatisch gebildet hat, um auf die Tabelle zuzugreifen.

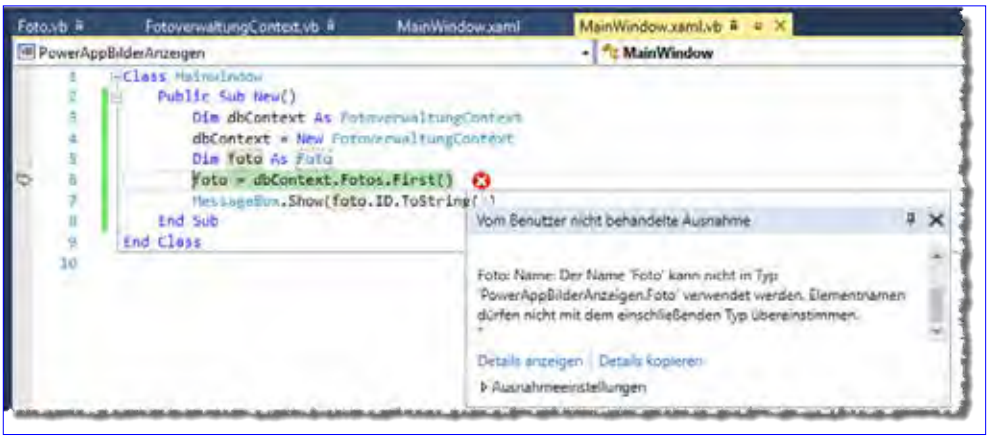


Bild 3: Fehler beim Zugriff auf die Tabelle

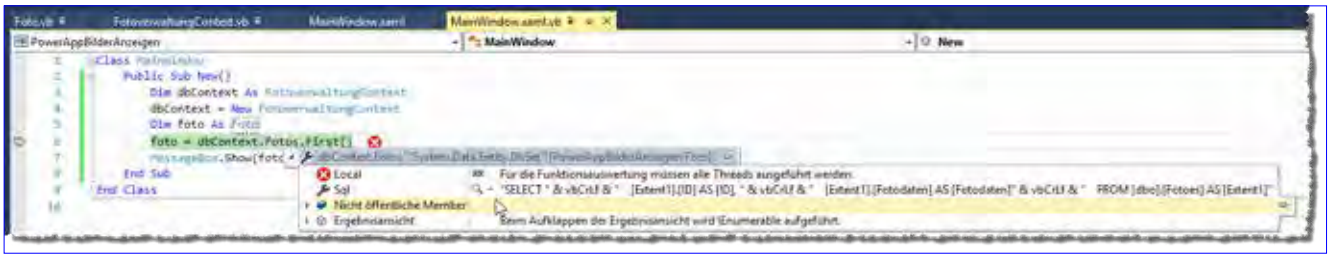


Bild 4: Der Zugriff erfolgt auf die nicht vorhandene Tabelle **Fotoes**

Damit Entity Framework erkennt, dass wir über die **DbSet**-Auflistung Fotos auf die Daten der Tabelle **tblFotos** zugreifen wollen, fügen wir der Methode **OnModelCreating** die folgende Anweisung hinzu:

```
Protected Overrides Sub OnModelCreating(ByVal modelBuilder As DbModelBuilder)
    modelBuilder.Entity(Of Foto)().ToTable("tblFotos")
End Sub
```

Daraufhin erhalten wir die Meldung aus Bild 5, die uns darauf hinweist, dass das Unterstützungsmodell geändert worden sei und wir mit einer Code First-Migration die Änderungen im Datenmodell in die Datenbank übertragen könnten. Was ist damit gemeint?

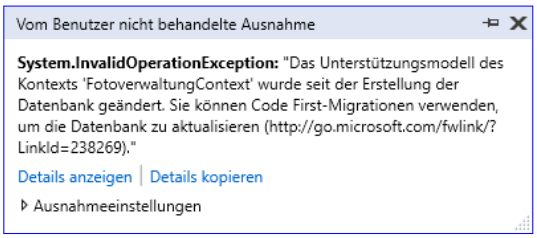


Bild 5: Unterstützungsmodell geändert?

Schauen wir uns nochmal den Inhalt des Objekts **Fotos** im Debug-Modus an, sehen wir, dass Entity Framework zwar nun auf die richtige Tabelle namens **tblFotos** zugreift (siehe Bild 6). Allerdings lautet der Name des Feldes nun **Fotodaten**. In der Tabelle **tblFotos** der Datenbank heißt es allerdings **Foto**. Entity Framework denkt also nun offensichtlich anhand des Unterschiedes zwischen dem in der Abfrage genannten Feldnamen **Fotodaten** und dem in der Tabelle vorgefundenen Feld **Foto**, dass der Benutzer das Entity Data Modell geändert hat

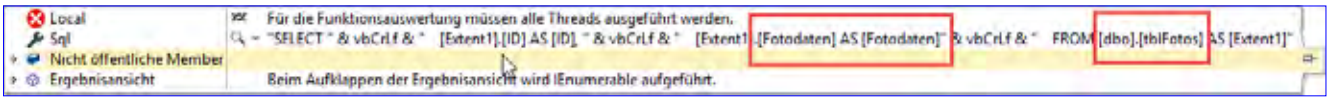


Bild 6: Die Anwendung versucht zwar, auf die richtige Tabelle zuzugreifen (**tblFotos**), aber noch nicht auf das richtige Feld.

und bietet eine Möglichkeit an, diese Änderung in die Datenbank zu übertragen. Das wollen wir allerdings nicht, sondern wir möchten das Mapping so anpassen, dass für die Eigenschaft **Fotodaten** der Entität **Foto** auf das Feld **Foto** der Tabelle **tblFotos** zugegriffen wird.

Also fügen wir noch einen weiteren Teil zur Methode **OnModelCreating** hinzu, mit der wir das Feld **Fotodaten** auf das Feld **Foto** der Tabelle **tblFotos** mappen. Das sieht dann wie folgt aus:

```
Protected Overrides Sub OnModelCreating(ByVal modelBuilder As DbModelBuilder)
    modelBuilder.Entity(Of Foto)().
        ToTable("tblFotos").
        Property(Function(t) t.Fotodaten).HasColumnName("Foto")
End Sub
```

Beim nächsten Start erhalten wir allerdings wieder die gleiche Meldung mit dem Hinweis auf den Einsatz der Code First-Migrationen. Wenn wir uns den Inhalt von **dbContect.Fotos** ansehen, finden wir allerdings folgende SQL-Anweisung vor:

```
"SELECT " & vbCrLf & "    [Extent1].[ID] AS [ID], " & vbCrLf & "    [Extent1].[Foto] AS [Foto]" & vbCrLf & "    FROM
[dbo].[tblFotos] AS [Extent1]"
```

Wenn wir diese **SELECT**-Anweisung im SQL Server Management Studio in einer neuen Abfrage für die hier verwendete Datenbank ausführen, erhalten wir allerdings das gewünschte Ergebnis (siehe Bild 7).

Unterschiede per Migration aufdecken

In den Artikeln **Datenbank-Initialisierung** und **Datenbank-Migration** haben wir die Migration von Code First-Datenbanken in das jeweilige Datenbanksystem beschrieben. Wie dort erläutert, werden wir nun die Migration aktivieren. Wir wollen allerdings keine Migration ausführen, sondern diese nur nutzen, um herauszufinden, welche Unterschiede zwischen dem Entity Data Model und dem Datenmodell der Datenbank zu dem aufgetretenen Fehler führen. Also öffnen Sie die Paket-Manager-Konsole und geben dort den folgenden Befehl ein:

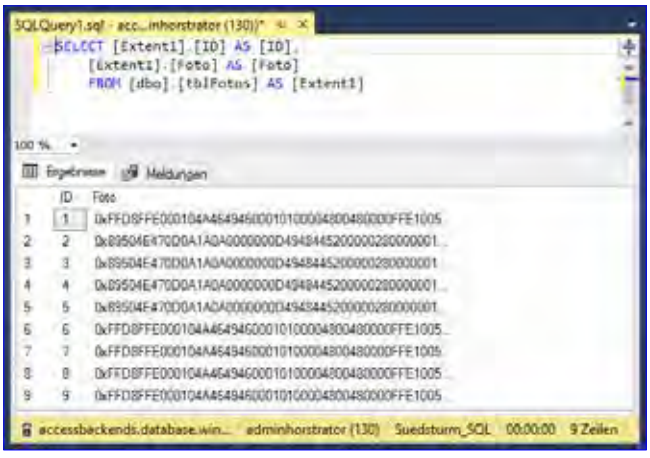


Bild 7: Die Abfrage funktioniert wie gewünscht.

```
enable-migrations
```

Dadurch wird der **Migrations**-Ordner angelegt, der auch direkt eine Klasse namens **...InitialCreate.vb** enthält. Das ist normalerweise nicht der Fall – üblicherweise müssen Sie

die erste Migration mit dem Befehl **add-migration <name>** erstellen.

Diese Klasse schauen wir uns nun an und finden sie wie in Bild 8 vor. Wie hier zu erkennen, will das Entity Framework eine Tabelle namens **Fotoes** erstellen. Das erklärt auch direkt das Problem: Mit den aktuellen (Standard-)Einstellungen erstellt Entity Framework aus den Namen der Entitätsklassen jeweils die Pluralform, es wird also **-es** oder **-s** an den Entitätsnamen angehängt. Aber auch hierfür gibt es noch eine Einstellung, die wir in der Methode **OnModelCreating** vornehmen können. Danach sieht die Methode wie folgt aus:

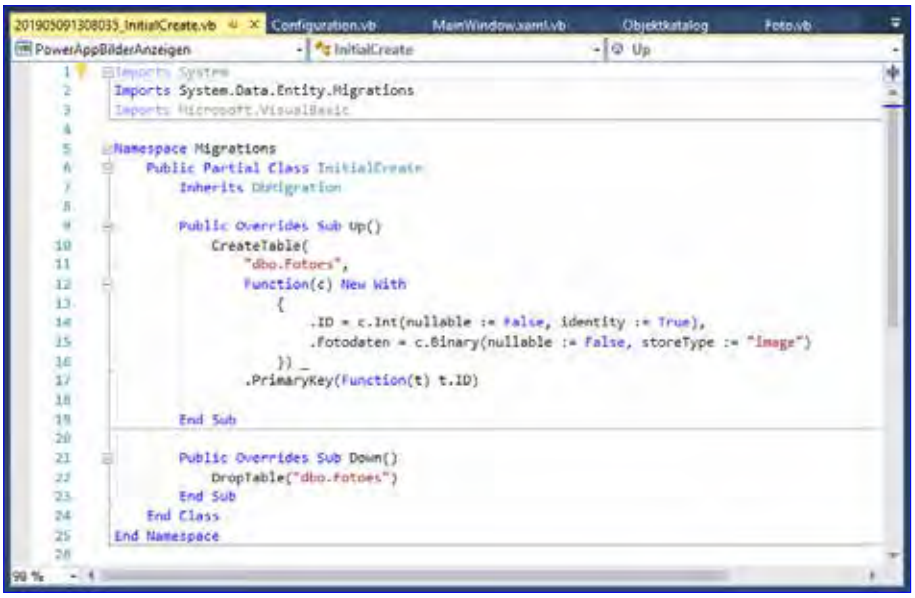


Bild 8: Die durchzuführenden Befehle, um das Entity Data Model in das Datenmodell zu migrieren

```
Protected Overrides Sub OnModelCreating(ByVal modelBuilder As DbModelBuilder)
    modelBuilder.Conventions.Remove(Of PluralizingTableNameConvention)()
    modelBuilder.Entity(Of Foto)().
        ToTable("tblFotos").
        Property(Function(t) t.Fotodaten).HasColumnName("Foto")
End Sub
```

Nach der Ergänzung erhalten wir allerdings nach wie vor die gleiche Fehlermeldung. Also löschen wir den Ordner **Migrations** und führen nochmals die Anweisung **enable-migrations** in der Paket-Manager-Konsole aus. Auch dies schafft jedoch keine Abhilfe. Entity Framework möchte nach wie vor eine Tabelle namens **Fotoes** erstellen. Aber auch das Erstellen einer Tabelle namens **Foto** wäre ja falsch. Wir wollen eine Tabelle namens **tblFotos** haben, die ja auch schon in der Datenbank vorhanden ist.

Wo aber kommt immer noch die Idee der **InitialCreate.vb**-Klasse her, dass wir eine Tabelle namens **Fotoes** in der Datenbank benötigen? Des Rätsels Lösung ist die Tabelle **_MigrationsHistory** in der Datenbank. Diese wurde wohl zwischendurch angelegt und enthält Informationen, die Entity Framework an dieser Stelle irritiert haben. Nachdem wir diese Tabelle in der Datenbank gelöscht haben, konnten wir ohne Probleme mit unserer Anwendung mit dem **DbSet** namens **Fotos** und der Klasse **Foto** auf die Daten der Tabelle **tblFotos** zugreifen.

Zusammenfassung der ersten Schritte

Die Aufgabe lautete, die Tabelle **tblFotos** mit den beiden Feldern **ID** und **Foto** über Entity Framework mit einem Entity Data Model zu verbinden, das eine Klasse namens Foto und eine **DbSet**-Auflistung namens **Fotos** aufweist und bei der wir auch noch den Namen des Feldes **Foto** in **Fotodaten** ändern mussten. Das gelingt, indem wir die folgenden einfachen Schritte durchführen:

EDM: Blättern in Datensätzen

Wenn Sie und Ihre Kunden die Arbeit mit der Detailansicht von Access-Formularen gewohnt sind, möchten Sie vielleicht auch in einem WPF-Fenster mit der Detailansicht der Daten einer Klasse blättern können. Dieser Artikel zeigt, wie Sie einem Detailformular die Daten einer Tabelle über die entsprechende DbSet-Auflistung zuweisen und wie Sie mit entsprechenden Schaltflächen in den Datensätzen blättern können.

Ziel der Erweiterung

Welche Möglichkeiten zum Blättern in den Datensätzen wollen wir dem Benutzer anbieten? Der Benutzer soll zum ersten, zum letzten, zum vorherigen und zum nächsten Datensatz blättern können. Außerdem wäre es hilfreich, wenn direkt ein neuer, leerer Datensatz angezeigt werden könnte.

Voraussetzungen

Wir bauen uns auf die Schnelle eine Beispielanwendung zusammen. Dazu nutzen wir die Techniken, die wir im Artikel **Detailformulare mit weiteren Steuerelementen** vorgestellt haben. Alternativ können Sie auch einfach das Beispielprojekt aus dem Download verwenden.

Die Schritte sehen in aller Kürze wie folgt aus:

- Neues Projekt auf Basis der Vorlage **Visual Basic\Windows Desktop\WPF-App** erstellen.
- Neues Element des Typs **ADO.NET Entity Data Model** namens **BestellverwaltungContext** mit der Vorlage **Leeres Code First-Modell** hinzufügen.
- In der Access-Datenbank **AccessZuWPFFormulare.accdb** aus dem Download die Prozedur **EDMErstellen** des Moduls **mdlEDM** aufrufen und den entstehenden Code aus der Zwischenablage nach den Anweisungen in den Kommentaren an den entsprechenden Stellen platzieren. Hier finden sich auch die Hinweise auf die Aufrufe der Befehle Enable-Migrations, Add-Migration und Update-Database, mit denen die Datenbank erstellt und gefüllt wird.
- Den XAML-Code für das gewünschte Detailformular, zum Beispiel **frmKundendetails**, mit dem folgenden Aufruf der Prozedur **FormularNachWPF** in die Zwischenablage kopieren: **FormularNachWPF "frmKundendetails", "tblKunden"**
- Den Inhalt der Zwischenablage in das Fenster **MainWindow.xaml** kopieren (oder in ein anderes, neues Fenster)
- Den VB-Code für die Code behind-Datei mit dem folgenden Aufruf der Prozedur **FormularNachWPF_CodeBehind** in die Zwischenablage kopieren: **FormularNachWPF_CodeBehind "frmKundendetails", "Kunde", "Kunden", "BestellverwaltungContext"**
- Den Inhalt der Zwischenablage in das Code behind-Modul **MainWindow.xaml.vb** oder das entsprechende andere Code behind-Modul kopieren.

Das Ergebnis nach dem Start der Anwendung sieht wie in Bild 1 aus. Hier wollen wir nun die Schaltflächen zum Blättern in den Datensätzen hinzufügen.

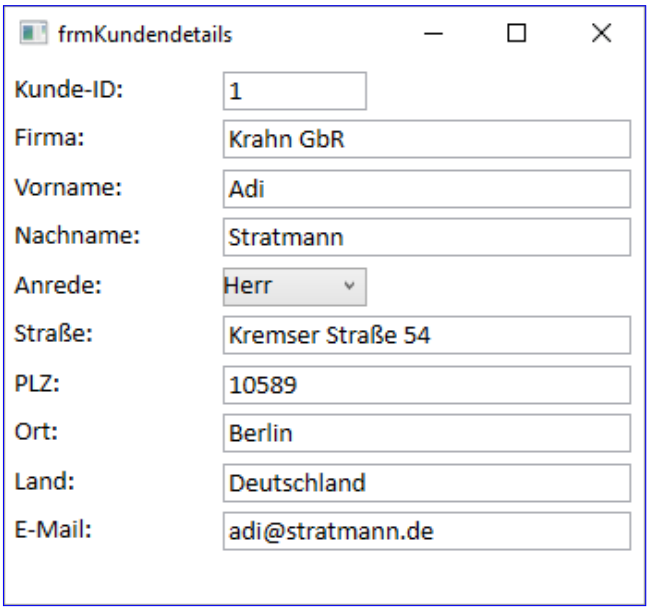


Bild 1: Fenster ohne Schaltflächen zum Blättern

Schaltflächen hinzufügen

Die Schaltflächen fügen wir in einem Element namens **StackPanel** hinzu, das ein Container für nebeneinander oder übereinander angeordnete Steuerelemente ist. Dieses Element füllen wir mit fünf **Button**-Elementen, deren Code wir wie folgt ergänzen:

```
<StackPanel Orientation="Horizontal"
HorizontalAlignment="Left" Height="28" Margin="0,273,0,0"
VerticalAlignment="Top" Width="339">
    <Button x:Name="btnErster" Content="&lt;&lt;"
Click="btnErster_Click" Width="25" Margin="2,2,2,2"/>
    <Button x:Name="btnVorheriger" Content="&lt;"
Click="btnVorheriger_Click" Width="25" Margin="2,2,2,2"/>
    <Button x:Name="btnNaechster" Content="&gt;" Click="btnNaechster_Click" Width="25" Margin="2,2,2,2"/>
    <Button x:Name="btnLetzter" Content="&gt;&gt;" Click="btnLetzter_Click" Width="25" Margin="2,2,2,2"/>
    <Button x:Name="btnNeu" Content="*" Click="btnNeu_Click" Width="25" Margin="2,2,2,2"/>
</StackPanel>
```

Dabei müssen die Kleiner- und Größer-Zeichen im Code in der Eigenschaft **Content** codiert werden, und zwar das Größer-Zeichen als **<** und das Kleiner-Zeichen als **>**. Für jede Schaltfläche haben wir ein Attribut namens **Click** angelegt, für das wir den Namen der beim Anklicken auszuführenden Ereignismethode angeben. Um die Eigenschaft schnell zu füllen, geben Sie das erste Anführungszeichen ein und wählen dann den IntelliSense-Eintrag **Neuer Ereignishandler** aus. Dadurch werden dann auch direkt die entsprechenden, noch leeren Ereignismethoden in der Code behind-Klasse erstellt.

Anzeigen des ersten Datensatzes

Damit beim Öffnen des Fensters direkt der erste Datensatz der Datenherkunft angezeigt wird, haben wir eine Konstruktor-Methode hinterlegt, welche die folgenden Befehle ausführt. Als Erstes füllen wir dabei die Variable **dbContext** mit einer neuen Instanz unserer Context-Klasse **BestellverwaltungContext**. Dann füllen wir die Auflistung **Kunden** mit einer **ObservableCollection**, die alle Einträge des **DbSet**-Objekts **Kunden** liefert. Schließlich füllen wir die Eigenschaft **Kunde** mit dem ersten Element der Auflistung **Kunden** (**Kunden.First()**) und füllen auch noch die Auflistung **Anreden**, welche die Datensätze für das **ComboBox**-Element **Anreden** enthält:

```
Public Sub New()
    InitializeComponent()
    dbContext = New BestellverwaltungContext
    Kunden = New ObservableCollection(Of Kunde)(dbContext.Kunden)
```

```
    Kunde = Kunden.First()
    Anreden = New List(Of Anrede)(dbContext.Anreden)
    DataContext = Me
End Sub
```

Ereignismethoden füllen

Damit zeigt das Fenster nun immer den ersten Datensatz an. Wir wollen aber mit den bereits angelegten Schaltflächen zwischen den Datensätzen navigieren können. Dazu schauen wir uns zunächst an, wie wir vom aktuellen Datensatz zum nächsten Datensatz gelangen, also zum zweiten Datensatz der **ObservableCollection** namens **Kunden**.

Das Problem ist, dass wir nicht durch eine ObservableCollection navigieren können wie etwa durch ein Recordset und seinen Methoden **MoveNext**, **MovePrevious** und so weiter. Wir können aber immerhin die Methode Item() mit dem gewünschten Index nutzen, um ein anderes Element zu referenzieren. Im folgenden Code-Beispiel weisen wir der Variablen **Kunde** etwa das zweite Element der Kundenliste zu:

```
Private Sub btnNaechster_Click(sender As Object, e As RoutedEventArgs)
    Kunde = Kunden.Item(1)
End Sub
```

Wenn wir das Projekt nun starten und auf die Schaltfläche **btnNaechster** klicken, wird allerdings immer noch der erste Datensatz angezeigt.

Der Grund ist schnell gefunden: Wir haben keinerlei Mechanismus implementiert, der dafür sorgen würde, dass Änderungen an dem in der Eigenschaft **Kunde** gespeicherten Element sich auf die Benutzeroberfläche auswirken beziehungsweise die Daten eines neu zugewiesenen **Kunde**-Elements angezeigt würden. Für diesen Fall ist die Schnittstelle **INotifyPropertyChanged** vorgesehen. Diese müssen wir in der Code behind-Klasse implementieren. Dazu fügen wir zunächst einen Verweis auf den Namespace **System.ComponentModel** hinzu:

```
Imports System.ComponentModel
```

Die Schnittstelle **INotifyPropertyChanged** müssen wir nun für die Klasse **frmKundendetails** implementieren. Dazu hängen wir in der Zeile unter der Definition der Klasse die folgende Zeile an:

```
Public Class frmKundendetails
    Implements INotifyPropertyChanged
    ...
End Class
```

Anschließend klicken Sie mit der rechten Maustaste auf **INotifyPropertyChanged**, wählen den Kontextmenü-Eintrag **Schnellaktionen und Refactorings...** aus und klicken dann auf **Schnittstelle implementieren** (siehe Bild 2). Daraufhin fügt Visual Studio die folgende Anweisung zur Klasse hinzu:

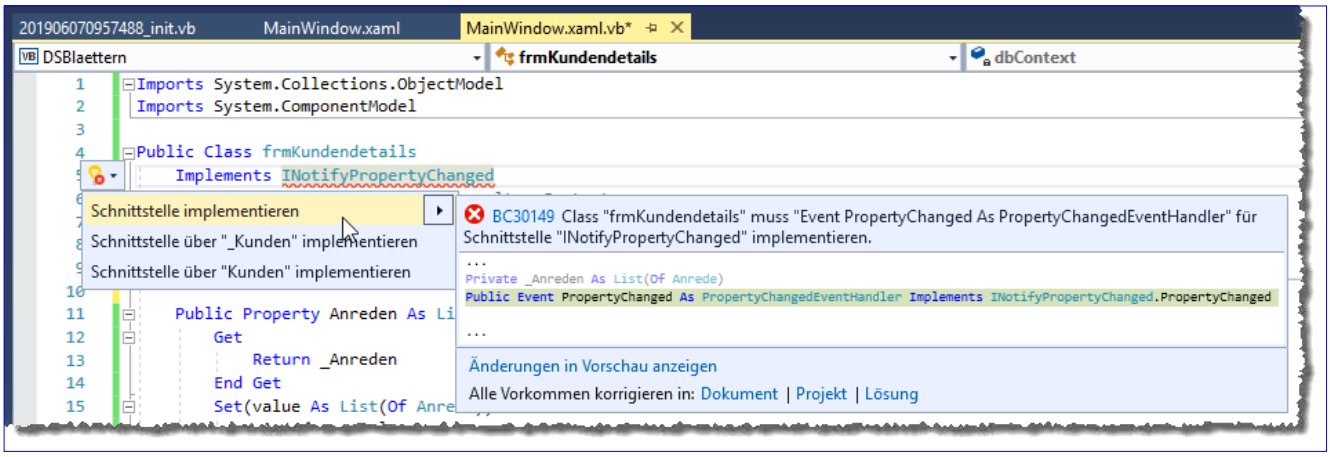


Bild 2: Implementieren der Schnittstelle **INotifyPropertyChanged**

```
Public Event PropertyChanged As PropertyChangedEventHandler Implements INotifyPropertyChanged.PropertyChanged
```

Nun müssen wir noch dafür sorgen, dass dieses Ereignis beim Ändern der Eigenschaft **Kunde** ausgelöst wird. Das erreichen wir durch Hinzufügen einer Zeile zur Definition der entsprechenden Property:

```
Public Property Kunde
    Get
        Return _Kunde
    End Get
    Set
        _Kunde = Value
        RaiseEvent PropertyChanged(Me, New PropertyChangedEventArgs("Kunde"))
    End Set
End Property
```

Wenn wir nun das Projekt neu starten und mit der Schaltfläche **btnNaechster** den Kunden wechseln wollen, wird dieser auch angezeigt.

Hinweis: Diese Änderung wurde bereits in den Code zum Erstellen der Code behind-Klasse von Access aus hinzugefügt.

Erster, vorheriger, nächster und letzter Datensatz

Damit können wir uns nun im Detail um die vier Schaltflächen **btnErster**, **btnVorheriger**, **btnNaechster** und **btnLetzter** kümmern. Den ersten Kunden finden wir leicht über die Methode **First** der **Kunden**-Auflistung. Also füllen wir die Methode **btnErster_Click** wie folgt:

```
Private Sub btnErster_Click(sender As Object, e As RoutedEventArgs)
    Kunde = Kunden.First
End Sub
```

Genauso leicht ist es, den letzten Datensatz anzusteuern. Dazu verwenden wir analog die **Last**-Methode:

```
Private Sub btnLetzter_Click(sender As Object, e As RoutedEventArgs)
    Kunde = Kunden.Last
End Sub
```

Für den vorherigen und nächsten Datensatz müssen wir ein wenig mehr investieren. Methoden wie **Previous** oder **Next** gibt die **ObservableCollection** nicht her. Wir wissen aber, dass wir einen beliebigen Eintrag mit der Eigenschaft **Item** und dem nullbasierten Indexwert ansteuern können. Wenn wir den Index des aktuellen Eintrags herausfinden können, brauchen wir nur noch eins hinzuzuzählen oder abzuziehen, um den Index des nächsten oder des vorherigen Datensatzes zu ermitteln. Den Index liefert die **IndexOf**-Eigenschaft der **Kunden**-Auflistung mit dem zu untersuchenden **Kunde**-Element als Parameter, also mit **Kunden.IndexOf(Kunde)**. Wenn wir diese Erkenntnisse kombinieren, erhalten wir die folgenden beiden Anweisungen für die Schaltflächen **btnVorheriger** und **btnNaechster**:

```
Private Sub btnVorheriger_Click(sender As Object, e As RoutedEventArgs)
    Kunde = Kunden.Item(Kunden.IndexOf(Kunde) - 1)
End Sub

Private Sub btnNaechster_Click(sender As Object, e As RoutedEventArgs)
    Kunde = Kunden.Item(Kunden.IndexOf(Kunde) + 1)
End Sub
```

Damit können wir nun wie gewünscht durch die Datensätze navigieren. Leider erhalten wir noch eine Fehlermeldung, wenn wir uns auf dem ersten Element befinden und die Schaltfläche **btnVorheriger** betätigen oder wenn wir uns auf dem letzten Element befinden und dann auf **btnNaechster** klicken. Der Grund ist, dass es vor dem ersten und hinter dem letzten Element keine weiteren Elemente mehr gibt. Dem können wir vorbeugen, indem wir die Schaltflächen **btnVorheriger** und **btnNaechster** deaktivieren, wenn aktuell das erste oder letzte Element angezeigt wird.

Dazu verwenden wir eine kleine Prozedur, die prüft, welchen Index der aktuell angezeigte Kunde aufweist. Dazu fragen wir die Eigenschaft **IndexOf** ab und speichern das Ergebnis in einer Variablen namens **lngIndex**. Danach aktivieren wir die Schaltfläche **btnVorheriger**, wenn **lngIndexID** nicht **0** ist und **btnNaechster**, wenn **lngIndexID** nicht der Anzahl der Elemente von **Kunden** entspricht:

```
Private Sub SchaltflaechenAktivieren()
    Dim lngIndex As Long
    lngIndex = Kunden.IndexOf(Kunde)
    btnVorheriger.IsEnabled = (Not lngIndex = 0)
    btnNaechster.IsEnabled = (Not lngIndex = Kunden.Count - 1)
End Sub
```

Diese Prozedur rufen wir in allein vier bisher beschriebenen Ereignismethoden wie folgt aus:

Validieren mit VB und EDM

Im Artikel EDM: Validieren von Entitäten mit IDataErrorInfo haben wir bereits erläutert, wie Sie einer auf dem Entity Framework basierenden Anwendung eine Validierung hinzufügen können. Im vorliegenden Artikel schauen wir uns an, wie Sie die Validierung nicht unter C#, sondern unter VB realisieren und welche Änderungen in den Methoden zum Migrieren einer Access-Anwendung nach WPF/EDM notwendig sind, um die Validierung vorzubereiten oder gegebenenfalls auch direkt umzusetzen.

Für die Programmierung einer Validierung in den Klassen des Entity Data Models und der mit WPF definierten Benutzeroberfläche sind einige Schritte auf den verschiedenen Ebenen erforderlich. Der erste ist, die Entitätsklassen mit der Implementierung der Schnittstelle **IDataErrorInfo** zu versehen, der zweite die Definition der Anzeige der Validierungsmeldungen in der Benutzeroberfläche – und das geschieht durch entsprechende WPF-Elemente.

IDataErrorInfo – wo implementieren?

Im oben genannten Artikel haben wir unter C# die Schnittstelle **IDataErrorInfo** für partielle Klassen implementiert. Das bedeutet, dass wir zwei gleichnamige Teilklassen programmiert haben, wobei wir in der einen die Definition der Properties und in der anderen die Implementierung der Schnittstelle **IDataErrorInfo** eingefügt haben. Warum haben wir das dort so gemacht? Weil wir den Code für die Entitätsklassen aus dem Datenmodell der Datenbank generiert haben, was bedeutet, dass wir bei Änderungen des Datenmodells auch die Entitätsklassen neu generieren mussten. Wenn wir dann die Implementierung der Schnittstelle **IDataErrorInfo** in der automatisch generierten Entitätsklasse programmiert hätten, wäre diese bei jeder Aktualisierung wieder überschrieben worden. Deshalb haben wir diese Definition direkt in eigenen, partiellen Klassen programmiert. Partielle Klasse bedeutet dabei, dass wir einfach eine weitere Klasse gleichen Namens mit dem Schlüsselwort **Partial** erstellt und dort die Elemente für die Implementierung der Schnittstelle hineingeschrieben haben. Die Klassenbezeichnungen der per Generator erzeugten Entitätsklassen wurden dabei praktischerweise bereits mit dem Schlüsselwort Partial ausgezeichnet – was keinen Nachteil bringt, auch wenn es keine weiteren partiellen Klassen mehr gibt.

Nun verwenden wir in der aktuell in diesem Magazin beschriebenen Vorgehensweise eine Access-Datenbank als Vorlage für die Erstellung eines Entity Data Models und auch für die Erstellung von Fenstern auf Basis der Formulare der Access-Datenbank. Wir könnten also theoretisch auch gleich die Implementierung der Schnittstelle **IDataErrorInfo** in die Entitätsklassen schreiben – und hier direkt die im Datenmodell festgelegten Restriktionen wie keine Nullwerte et cetera eintragen. Allerdings kann es auch hier sein, dass Sie die Implementierung der Schnittstelle **IDataErrorInfo** selbst anpassen wollen. Also werden wir auch hier mit partiellen Klassen arbeiten und per Parameter beim Erstellen des Entity Data Models auf Basis der Tabellen der Access-Datenbank die Möglichkeit bieten, entweder nur das Entity Data Model zu erstellen oder auch noch die partiellen Klassen mit den Validierungsfunktionen neu zu generieren. Zunächst schauen wir uns allerdings an, wie die partiellen Klassen unter VB überhaupt aussehen sollen.

Partielle Klasse mit IDataErrorInfo erstellen

Das Anlegen der zusätzlichen partiellen Klasse etwa für die Klasse **Kunde** erfordert zwei Schritte. Als Erstes fügen wir der eigentlichen Klasse das Schlüsselwort **Partial** als erstes Schlüsselwort noch vor **Public** hinzu:

```
<Table("Kunden")>
Partial Public Class Kunde
    ...
End Class
```

Danach können wir die zweite partielle Klasse erstellen:

```
Partial Public Class Kunde
    ...
End Class
```

Für diese Klasse wollen wir nun die Schnittstelle **IDataErrorInfo** implementieren. Dazu benötigen wir zunächst die Bibliothek System.**ComponentModel**, die wir per **Imports**-Anweisung einbinden:

```
Imports System.ComponentModel
```

Danach fügen Sie in der Zeile unter der Klassendefinition die Anweisung zur Implementierung der Schnittstelle hinzu:

```
Partial Public Class Kunde
    Implements IDataErrorInfo
    ...
```

Wenn Sie mit der rechten Maustaste auf **IDataErrInfo** klicken, erscheint der Kontextmenü-Eintrag **Schnellaktionen und Refactorings...**, den Sie anklicken und danach den nun erscheinenden Befehl **Schnittstelle implementieren** anklicken (siehe Bild 1).

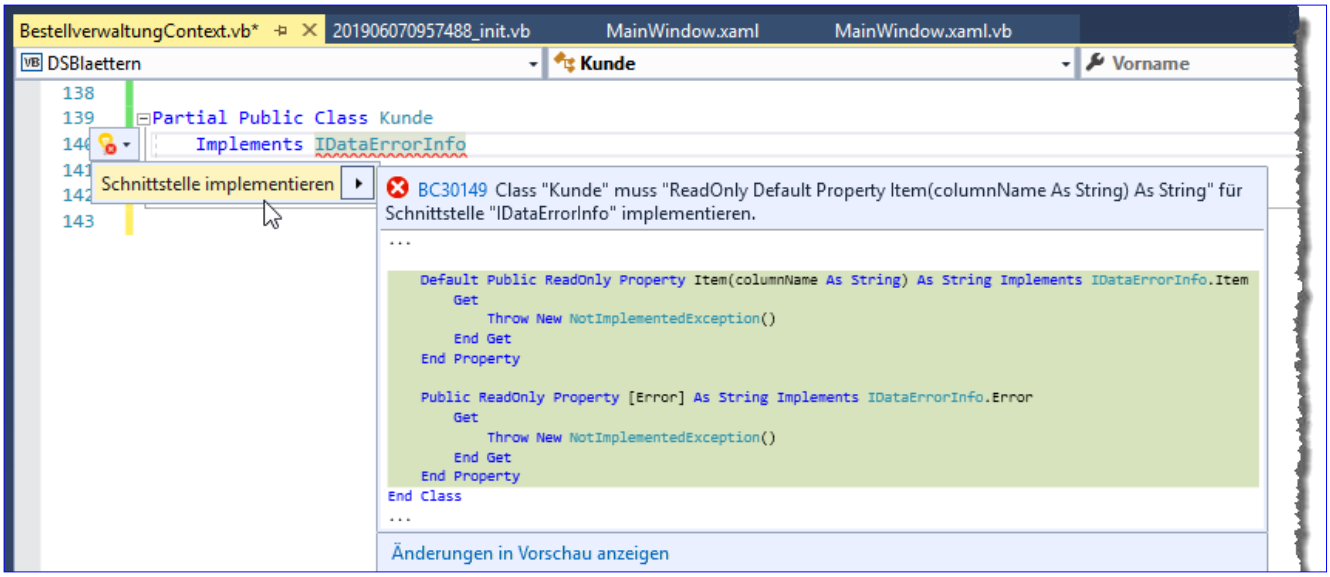


Bild 1: Automatisches Implementieren der Schnittstelle

Das füllt die Klasse mit den beiden Properties **Item** und **[Error]**:

```
Partial Public Class Kunde
    Implements IDataErrorInfo

    Default Public ReadOnly Property Item(columnName As String) As String Implements IDataErrorInfo.Item
        Get
            Throw New NotImplementedException()
        End Get
    End Property

    Public ReadOnly Property [Error] As String Implements IDataErrorInfo.Error
        Get
            Throw New NotImplementedException()
        End Get
    End Property
End Class
```

Uns interessiert dabei speziell die **Property** namens **Item**. Diese liefert mit dem Parameter **columnName** den Namen des Feldes, für das eine Validierung durchgeführt werden soll. Um eine Validierung für das Feld **Vorname** durchzuführen, die prüfen soll, ob das Feld **Vorname** leer ist, fügen Sie die folgende **Select Case**-Bedingung mit dem **Case**-Zweig für den Feldnamen **Vorname** hinzu.

Die Prüfung stellt mit der Methode **IsNullOrEmpty** fest, ob **Vorname** leer ist. In diesem Fall wird der Rückgabeparameter **strErrorMessage** mit einer entsprechenden Fehlermeldung gefüllt:

```
Default Public ReadOnly Property Item(columnName As String) As String Implements IDataErrorInfo.Item
    Get
        Dim strErrorMessage As String = ""
        Select Case columnName
            Case "Vorname"
                If (String.IsNullOrEmpty(Vorname)) Then
                    strErrorMessage = "Bitte geben Sie einen Vornamen ein."
                End If
        End Select
        Return strErrorMessage
    End Get
End Property
```

Nun stellt sich nur noch die Frage, wodurch diese Methode aufgerufen wird und wie wir die Fehlermeldung aus **strErrorMessage** verarbeiten.

Auslösen der Validierung

Dass für das Feld **Vorname** eine Validierung ausgelöst wird, legen wir in der Definition des an dieses Feld gebundenen Steuerelements fest, in diesem Fall des Textfeldes **txtVorname**. Hier erweitern wir den Inhalt der Bindung für das Attribut **Text** um **ValidatesOnDataErrors=True**:

```
<TextBox x:Name="txtVorname" Text="{Binding Kunde.Vorname, ValidatesOnDataErrors=True}" Height="21" Margin="118,58,0,0"
Width="221"/>
```

Damit allein wird nun bereits die Validierung ausgelöst, das heißt, wenn das Feld **Vorname** angezeigt wird, ruft dies auch die Methode **Item** der Implementierung von **IDataErrorInfo** auf. Das Ergebnis ist allerdings noch keine Anzeige der Meldung, sondern zunächst nur das rote Einrahmen des Feldes, das nicht erfolgreich validiert werden konnte (siehe Bild 2).

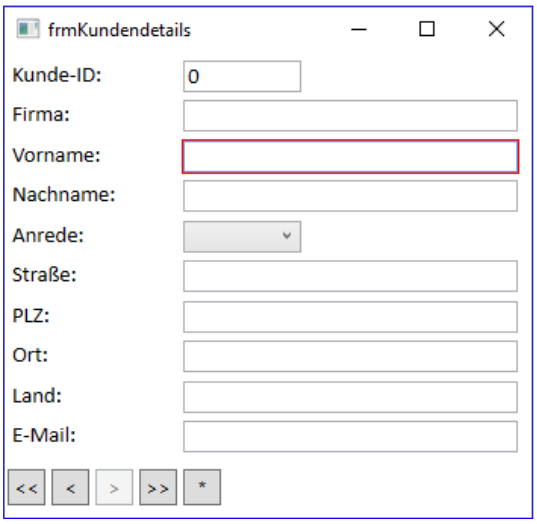


Bild 2: Die Validierung funktioniert bereits.

Wir wollen allerdings noch konkretere Hinweise auf den Grund für die farbige Markierung liefern, damit der Benutzer weiß, was zu tun ist. Also verwenden wir den Setter, den wir auch schon im oben genannten Artikel genutzt haben, um die Fehlermeldung an entsprechender Stelle unterzubringen. Dazu erweitern wir den Code für das **Style**-Element in der **.xaml**-Datei wie folgt:

```
<Style TargetType="{x:Type TextBox}">
    ...
    <Setter Property="Validation.ErrorTemplate">
        <Setter.Value>
            <ControlTemplate>
                <DockPanel>
                    <Border Background="Red" DockPanel.Dock="right" Margin="5,0,0,0" Width="20" Height="20"
                            CornerRadius="10" ToolTip="{Binding ElementName=customAdorner,
                            Path=AdornedElement.(Validation.Errors)[0].ErrorContent}">
                        <TextBlock Text="!" VerticalAlignment="center" HorizontalAlignment="center"
                            FontWeight="Bold" Foreground="white" />
                    </Border>
                    <AdornedElementPlaceholder Name="customAdorner" VerticalAlignment="Center" >
                        <Border BorderBrush="red" BorderThickness="1" />
                    </AdornedElementPlaceholder>
                </DockPanel>
            </ControlTemplate>
        </Setter.Value>
    </Setter>
</Style>
```

Danach erscheint die Meldung, die wir für das Feld **Vorname** definiert haben, wie in Bild 3.

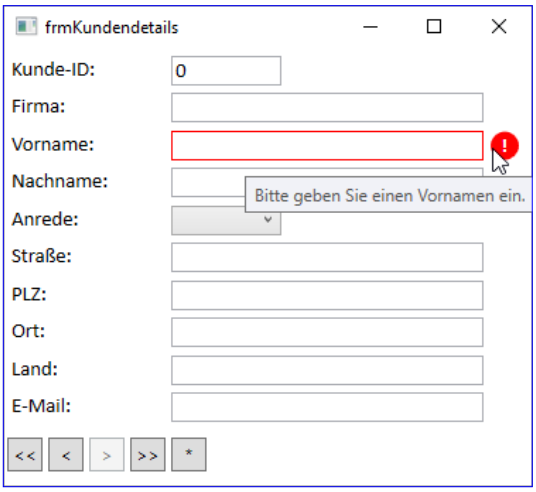


Bild 3: Validierung mit Hinweismeldung

Validierung von Kombinationsfeldern

Für die Validierung des Kombinationsfeldes **Anrede** fügen Sie in der **.xaml**-Datei für die Definition des **Style**-Elements für den Typ **ComboBox** genau die gleiche Property namens **Validation.ErrorTemplate** wie für den Typ **TextBox** hinzu:

```
<Style TargetType="{x:Type ComboBox}">
    ...
    <Setter Property="Validation.ErrorTemplate">
    ...
    </Setter>
</Style>
```

Außerdem passen Sie in der Definition des **ComboBox**-Elements das Attribut **SelectedItem** so an, dass die Bindung mit der Option **ValidatesOnDataErrors=True** erfolgt:

```
<ComboBox x:Name="cboAnredeID" Padding="0" Height="21" Margin="118,111,0,0" Width="78"
    ItemsSource = "{Binding Anreden}"
    SelectedItem = "{Binding Kunde.Anrede, ValidatesOnDataErrors=True}"
    ...
</ComboBox>
```

Schließlich erweitern wir die Implementierung der Schnittstelle **IDataErrorInfo** noch um einen entsprechenden **Case**-Zweig. Hier verwenden wir allerdings die **IsNothing**-Funktion, um zu prüfen, ob für die Eigenschaft **Anrede** eines der Elemente der Auflistung **Anreden** zugewiesen wurde:

```
Default Public ReadOnly Property Item(columnName As String) As String Implements IDataErrorInfo.Item
    Get
        Dim strErrorMessage As String = ""
        Select Case columnName
            ...
            Case "Anrede"
                If (IsNothing(Anrede)) Then
                    strErrorMessage = "Bitte wählen Sie eine Anrede aus."
                End If
        End Select
        Return strErrorMessage
    End Get
```

```
End Property
```

Verlassen des Datensatzes ohne erfolgreiche Validierung unterbinden

Nun müssen wir noch sicherstellen, dass der Benutzer den Datensatz nicht verlassen kann, ohne dass alle erforderlichen Daten eingegeben wurden. Dazu haben wir im oben genannten Artikel eine Technik kennengelernt, die auf den Ergebnissen unserer Validierung aufbaut und davon abhängig Steuerelemente aktiviert oder deaktiviert. Dabei haben wir für die betroffene Schaltfläche **DataTrigger**-Elemente als **Style**-Elemente festgelegt und für jeden Trigger eine Bedingung formuliert, unter der eine bestimmte Eigenschaft auf einen Wert eingestellt werden soll – in diesem Fall etwa der Wert der Validierung eines oder mehrerer gebundener Steuerelemente und damit verbunden die Eigenschaft **IsEnabled** der zu deaktivierenden Schaltfläche.

Daraus leiten wir für unsere Steuerelemente, hier etwa für die Schaltfläche **btnErster**, die entsprechende **Style**-Definition ab. Die **Style.Triggers**-Auflistung nimmt dabei mehrere **DataTrigger**-Elemente auf. Diese referenzieren jeweils die Eigenschaft **Validation.HasError** eines der Steuerelemente, in der nachfolgenden verkürzten Form etwa die Textfelder **txtFirma** oder **txtEMail**. Hat **Validation.HasError** den Wert **True**, wird der eingefasste **Setter** aktiviert, der hier den Wert der Eigenschaft **IsEnabled** auf **True** einstellt:

```
<Button x:Name="btnErster" Content="&lt;&lt;" Click="btnErster_Click" Width="25" Margin="2,2,2,2">
    <Button.Style>
        <Style TargetType="{x:Type Button}">
            <Style.Triggers>
                <DataTrigger Binding="{Binding ElementName=txtFirma, Path=(Validation.HasError)}" Value="True">
                    <Setter Property="IsEnabled" Value="False"></Setter>
                </DataTrigger>
                ...
                <DataTrigger Binding="{Binding ElementName=txtEMail, Path=(Validation.HasError)}" Value="True">
                    <Setter Property="IsEnabled" Value="False"></Setter>
                </DataTrigger>
            </Style.Triggers>
        </Style>
    </Button.Style>
</Button>
```

Diesen Code müssten wir für jede betroffene Schaltfläche festlegen, und zwar mit **DataTrigger**-Elementen für alle zu validierenden Steuerelemente. Da der Benutzer für ein nicht erfolgreich validiertes Element keine der Schaltflächen **btnErster**, **btnVorheriger**, **btnNaechster**, **btnLetzter** und **btnNeu** anklicken können soll, benötigen wir diesen Code also fünf Mal. Das ist schon eine Menge Code, den wir vermutlich noch kürzer fassen können. Vorher schauen wir uns allerdings an, ob das Fenster nun wie gewünscht arbeitet. Wie Bild 4 zeigt, ist das nicht der Fall.

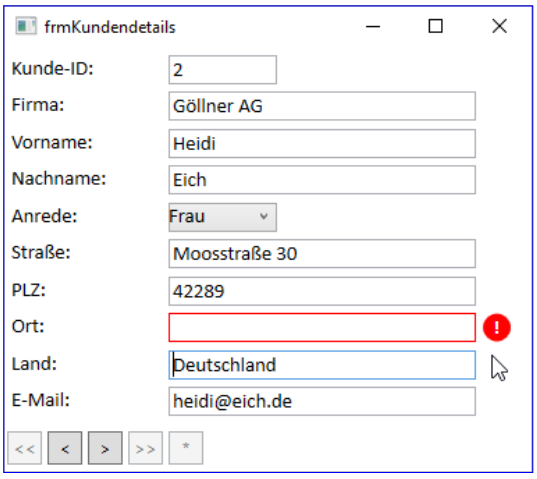


Bild 4: Die Schaltflächen **btnVorheriger** und **btnNaechster** werden nicht deaktiviert.

Entity Framework: Der ChangeTracker

Wenn Sie Daten etwa aus den Tabellen einer Datenbank in ein Entity Data Model geladen haben, finden Sie dort einige Funktionen für den Umgang mit den enthaltenen Daten vor. Ein sehr wichtiges Element ist dabei der ChangeTracker. Auch wenn Sie mit der SaveChanges-Methode automatisch alle Änderungen im Entity Data Model erkennen und diese in die Datenbank übertragen können, so treten doch Fälle auf, in denen Sie zuvor prüfen wollen, welche Änderungen überhaupt im Entity Data Model vorgenommen wurden – und ob diese in die Datenbank übernommen oder vielleicht sogar verworfen werden sollen.

Wenn Sie eine Anwendung bauen, wie wir sie in den letzten Ausgaben verwendet haben, dann bezieht diese ihre Daten aus den Tabellen eines SQL Servers und stellt diese in einem Entity Data Model bereit. Die Elemente des Entity Data Models binden Sie dann an die Benutzeroberfläche und die dort befindlichen Steuerelemente.

Nun können nach dem Anzeigen der Daten in einem Bearbeitungsfenster verschiedene Fälle auftreten: Sie ändern ein oder mehrere Felder eines Datensatzes, fügen einen neuen Datensatz hinzu oder löschen einen Datensatz. Diese Änderungen liegen dann im Entity Data Model vor, aber werden nicht automatisch in die Datenbank übertragen. Das erledigen Sie erst, wenn Sie die **SaveChanges**-Methode des Datenbankkontextes aufrufen. **SaveChanges** ist effizient, denn es prüft, ob Änderungen vorliegen und speichert nur die geänderten Elemente in der Datenbank.

Was aber geschieht, wenn Sie beispielsweise erfahren wollen, ob an einem bestimmten Datensatz Änderungen vorgenommen wurden? Für ein Fenster mit Minimalanforderungen benötigen Sie diese Information nicht, denn Sie können die Änderungen ja einfach in der Datenbank speichern. Wenn Sie dem Benutzer jedoch ein paar weitere Features wie etwa eine Schaltfläche zum Verwerfen der Änderungen zur Verfügung stellen wollen, kann es schon interessant sein, ob der Benutzer überhaupt schon Änderungen am aktuellen Datensatz vorgenommen hat. Dabei könnten Sie die Information, ob der Datensatz bereits geändert wurde, etwa dazu nutzen, die **Abbrechen**- oder **Verwerfen**-Schaltfläche nur zu aktivieren, wenn der Benutzer bereits Änderungen vorgenommen hat – oder wenn er soeben einen neuen Datensatz angelegt hat, der ja mitunter auch verworfen werden könnte.

Und hier kommt der **ChangeTracker** von Entity Framework ins Spiel. Dabei handelt es sich um eine Klasse, die zum Beispiel den Zugriff auf alle Elemente mit einem bestimmten Status zulässt – beispielsweise geändert oder nicht geändert. In diesem Artikel schauen wir uns nun an, wie Sie mit dem **ChangeTracker** von Entity Framework arbeiten können.

Beispielprojekt

Als Beispielprojekt verwenden wir ein Projekt auf Basis der Vorlage **Visual Basic\Windows Desktop\WPF-App** namens **ChangeTracker**. Dieser haben wir ein Entity Data Model mit der Methode aus der Artikelreihe **Von Access zu Entity Framework** hinzugefügt und eine SQL Server-Datenbank damit erstellt. Um die SQL Server-Datenbank auf ihrem Rechner zu erstellen, rufen Sie die Anweisung **Update-Database** in der Paket-Manager-Konsole auf. Dadurch wird eine Datenbank auf Basis der

in der Datei **App.config** angegebenen Verbindungszeichenfolge erstellt, die Sie gegebenenfalls noch anpassen können. Die verschiedenen Beispielcodes der folgenden Abschnitte finden Sie hinter den Ereignismethoden der Schaltflächen des Fensters **MainWindow.xaml**.

Die ChangeTracker-Klasse

Die **ChangeTracker**-Klasse referenzieren über die gleichnamige Eigenschaft der Datenbankkontext-Klasse, die in unserem Beispielen meist **dbContext** genannt wird. Mit dieser Klasse können Sie den Zustand jedes der Elemente des aktuellen Datenbankkontexts nachverfolgen. Jeder der Einträge hat einen Wert für die Eigenschaft **EntityState**. Die möglichen Werte lauten:

- **Added**: Hinzugefügtes Element, das noch nicht in der Datenbank existiert
- **Modified**: Geändertes Element, dessen Änderungen noch nicht in die Datenbank übertragen wurden
- **Deleted**: Gelöschtes Element, das noch in der Datenbank existiert
- **Unchanged**: Nicht geändertes Element
- **Detached**: Nicht verbundenes, also nicht zu einem **DbSet** gehörendes Element. Dieses wird auch nicht vom ChangeTracker verfolgt.

Beispiele zur ChangeTracker-Klasse

Wir schauen uns die verschiedenen Zustände an einfachen Beispielen an. Dazu fügen wir der Klasse **MainWindow.xaml.vb** zunächst die folgenden Namespaces hinzu:

```
Imports System.Data.Entity
Imports System.Data.Entity.Infrastructure
```

Danach erstellen wir die folgende Ereignismethode für die erste Schaltfläche **btnUnveraendert**. Hier füllen wir die Variable **dbContext** mit einer neuen Instanz des Datenbankkontextes und füllen das Objekt **Kunde** des Typs **Kunde** mit dem ersten Eintrag der Liste **Kunden** des Datenbankkontextes:

```
Private Sub btnUnveraendert_Click(sender As Object, e As RoutedEventArgs)
    Dim dbContext As BestellverwaltungContext
    Dim Element As EntityEntry
    Dim Kunde As Kunde
    dbContext = New BestellverwaltungContext
    Kunde = dbContext.Kunden.First()
```

Danach folgt der erste Zugriff auf den **ChangeTracker**. Dabei durchlaufen wir dessen **Entries**-Auflistung und referenzieren das aktuelle Element jeweils mit der Variablen **Element**, die übrigens den Typ **EntityEntry** aufweist (dafür auch der Verweis auf den Namespace). Anschließend geben wir für jeden Eintrag zwei Informationen aus – den Namen des Elementtyps und den

Zustand. Um den Namen des Elementtyps zu erhalten, referenzieren wir zunächst die in **Element** enthaltene Entität mit der Eigenschaft **Entity** und ermitteln den Namen des Typs mit **GetType().Name**. Den Zustand dieses Eintrags liefert die Eigenschaft **State** des **EntityEntry**-Elements:

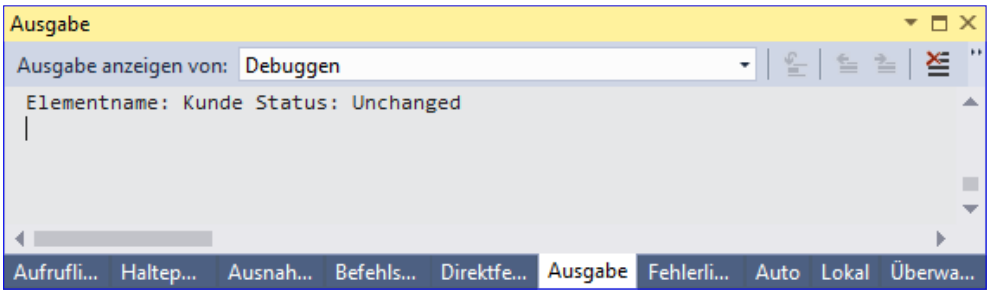


Bild 1: Ausgabe von Typ und Status einer Entität

```
    For Each Element In dbContext.ChangeTracker.Entries
        Debug.Print("Elementname: " + Element.Entity.GetType().Name + " Status: " + Element.State.ToString())
    Next
End Sub
```

Das Ergebnis sieht schließlich wie in Bild 1 aus. Der Typ **Kunde** wird korrekt erkannt und auch der Status entspricht mit **Unchanged** den Erwartungen.

ChangeTracker verfolgt alle Elementtypen

Wir fügen eine Variable für ein Element eines weiteren Elementtypen namens **Anrede** hinzu und lesen auch hier den ersten Eintrag aus – in diesem Fall aus der Liste **Anreden**:

```
Dim Anrede As Anrede
...
Anrede = dbContext.Anreden.First()
```

Die Ausgabe im Ausgabefenster sieht nach dem erneuten Start der Anwendung und dem Betätigen der Schaltfläche wie folgt aus:

```
Elementname: Kunde Status: Unchanged
Elementname: Anrede Status: Unchanged
```

Wir sehen also, dass die **Entries**-Auflistung der **ChangeTracker**-Klasse Elemente aller Typen umfasst – sie müssen nur in den Speicher geladen sein.

Ein Element löschen

Im nächsten Beispiel wollen wir untersuchen, was beim Löschen eines Eintrags geschieht. Dazu legen wir eine neue Schaltfläche namens **btnGeloeschterEintrag** hinzu, für deren **Click**-Ereignis wir die folgende Ereignismethode hinterlegen:

```
Private Sub btnGeloeschterEintrag_Click(sender As Object, e As RoutedEventArgs)
    Dim dbContext As BestellverwaltungContext
```

```
    Dim Bestellposition As Bestellposition
    Dim Element As DbEntityEntry
    dbContext = New BestellverwaltungContext
    Bestellposition = dbContext.Bestellpositionen.First()
    dbContext.Bestellpositionen.Remove(Bestellposition)
    For Each Element In dbContext.ChangeTracker.Entries
        MessageBox.Show("Elementname: " + Element.Entity.GetType().Name + vbCrLf + " Status: " _
            + Element.State.ToString())
    Next
End Sub
```

Hier laden wir nun das erste Element der Auflistung **Bestellpositionen**, da wir diese ohne Probleme löschen können – die Einträge haben keine Beziehungen, deren Restriktion ein Löschen verhindern könnte. Dann löschen wir dieses Element mit der **Remove**-Methode aus der Auflistung **Bestellpositionen**. Anschließend rufen wir wieder die gleiche **For Each**-Schleife über die **Entries**-Auflistung. Diesmal liefert die **State**-Eigenschaft den Wert **Deleted**.

Ein Element hinzufügen

Nun erstellen wir ein neues **Kunde**-Element, füllen seine Eigenschaften und fügen das **Kunde**-Element mit der **Add**-Methode zur Auflistung **Kunden** hinzu:

```
Kunde = New Kunde
With Kunde
    .Vorname = "André"
    .Nachname = "Minhorst"
    .AnredeID = 1
    .Strasse = "Borkhofer Str. 17"
    .PLZ = "47137"
    .Ort = "Duisburg"
    .Land = "Deutschland"
    .EMail = "andre@minhorst.com"
End With
dbContext.Kunden.Add(Kunde)
Element = dbContext.ChangeTracker.Entries.First
MessageBox.Show("Elementname: " + Element.Entity.GetType().Name + vbCrLf + " Status: " + Element.State.ToString())
```

Hier lautet der Wert von **State** nun **Added**.

Ein Element ändern

Wir legen noch eine neue Schaltfläche an, mit der wir ein bestehendes Element ändern. Dazu weisen wir dem Feld **Vorname** des **Kunde**-Elements einen anderen Wert zu. Da wir wissen, dass wir gerade nur einen Eintrag geändert haben, durchlaufen wir keine Schleife mehr, sondern greifen über die **First**-Eigenschaft auf das geänderte Element zu:

```
Kunde = dbContext.Kunden.First()
Kunde.Vorname = "André"
Element = dbContext.ChangeTracker.Entries.First
MessageBox.Show("Elementname: " + Element.Entity.GetType().Name + vbCrLf + " Status: " + Element.State.ToString())
```

In diesem Fall liefert die Eigenschaft **State** den Wert **Modified**.

Nicht verbundene Elemente

Es kann auch sein, dass Sie ein Element neu erstellen, aber dieses nicht mit der **Add**-Methode zur Auflistung des entsprechenden Elementtyps hinzufügen. Um zu untersuchen, welchen Zustand dieses Element hat und ob es vom **ChangeTracker** erfasst wird, müssen wir hier hier einen anderen Weg gehen. Wir finden nämlich, wie die zweite **MessageBox** zeigt, keinen Eintrag in der **ChangeTracker**-Liste **Entries** vor – das Ergebnis der **Count**-Eigenschaft ist **0**. Um ein **DbEntityEntry**-Element auf Basis des neuen **Kunde**-Objekts zu erstellen, greifen wir nun nicht auf die **Entries**-Liste der **ChangeTracker**-Klasse zu, sondern auf die **Entry**-Eigenschaft von **dbContext** für das **Kunde**-Objekt. Für dieses erhalten wir dann als **State** den Wert **Detached**, also nicht verbunden:

```
Dim dbContext As BestellverwaltungContext
Dim Kunde As Kunde
Dim Element As DbEntityEntry
dbContext = New BestellverwaltungContext
Kunde = New Kunde
With Kunde
    .Vorname = "André"
    ...
End With
Element = dbContext.Entry(Kunde)
MessageBox.Show("Elementname: " + Element.Entity.GetType().Name + vbCrLf + " Status: " + Element.State.ToString())
MessageBox.Show("Anzahl Elemente Changetracker: " + dbContext.ChangeTracker.Entries.Count().ToString())
```

Unterschied zwischen Entry(Element) und ChangeTracker.Entries

In den vorherigen Beispielen haben Sie gelernt, dass wir das **DbEntityEntry**-Objekt zu einem Element des Entity Data Models sowohl über die Eigenschaft **Entry** von **dbContext** holen können als auch über die Einträge der Auflistung **Entries** der **ChangeTracker**-Klasse. Welche der beiden Möglichkeiten Sie nutzen, hängt vom Anwendungszweck ab. Wenn Sie etwa alle Elemente mit bestimmten Eigenschaften in einer Schleife durchlaufen wollen, nutzen Sie **Entries**. Wenn Sie vorher schon einen Verweis auf das zu untersuchende Element haben, können Sie dieses auch direkt etwa über **Entry(Kunde)** referenzieren. Die Werte der Eigenschaften wie etwa **State** bleiben davon unberührt.

Geänderte Elemente untersuchen

Wir haben bereits gesehen, dass geänderte Elemente mit dem **State**-Wert **Modified** versehen werden. Nun interessiert uns noch, wie wir herausfinden, welche Eigenschaft der Elemente geändert wurden. Dazu ermitteln wir wieder das erste Element der **Kunden**-Auflistung und speichern es in der Variablen **Kunde**. Wir ändern den Wert des Feldes **Vorname** und weisen das

Änderungen erkennen und verwerfen

Unter Access können Sie beispielsweise mit der Escape-Taste die aktuellen ungespeicherten Änderungen an einem Datensatz verwerfen. Das Formular zeigt dann direkt die Daten an, die beim letzten Speichern in den gebundenen Feldern enthalten waren. Dieses Verhalten wollen wir auch für Fenster abbilden, die an ein Element gebunden sind und die Daten in einer Detailansicht anzeigen. Wir wollen erkennen, ob der Benutzer Änderungen an einem Datensatz vorgenommen hat und in Abhängigkeit davon eine Rückgängig-Schaltfläche aktivieren oder deaktivieren. Durch einen Klick auf die Schaltfläche sollen die vorherigen Werte wiederhergestellt werden. Wie das gelingt, zeigt der vorliegende Beitrag.

Ein WPF-Fenster bietet nicht automatisch eine **Dirty**-Eigenschaft wie ein gebundenes Access-Formular. Eine solche Funktion müssen wir selbst zu einem Fenster hinzufügen. Das Ergebnis dieses Artikels soll etwa wie in Bild 1 aussehen. Das Ändern eines Eintrags und das Verlassen dieses Eintrags sollen dafür sorgen, dass die **Rückgängig**-Schaltfläche aktiviert wird.

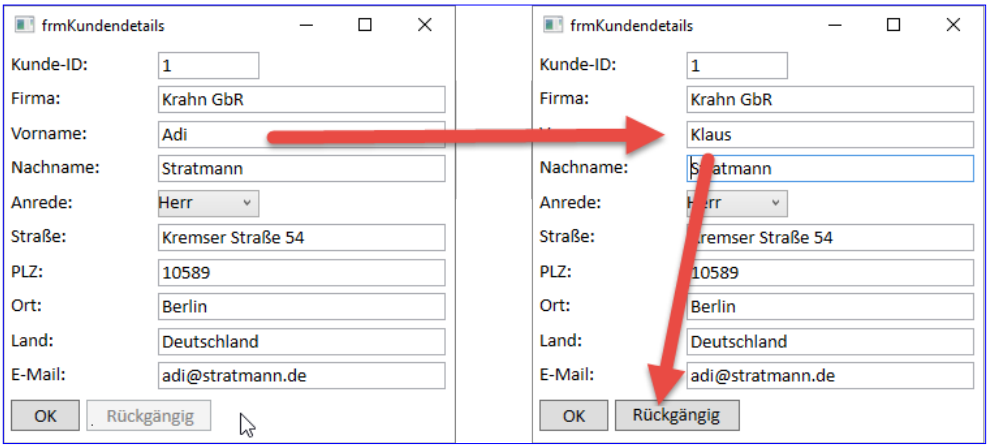


Bild 1: Aktivieren der **Abbrechen**-Schaltfläche nach einer Änderung

Betätigt der Benutzer dann die **Rückgängig**-Schaltfläche, soll der Datensatz auf die Version beim Laden zurückgesetzt werden (siehe Bild 2).

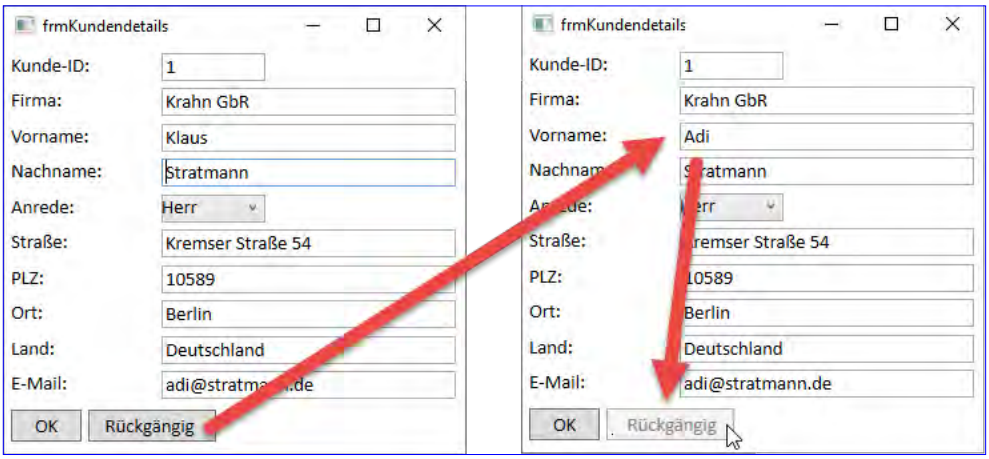


Bild 2: Zurücksetzen des Datensatzes nach Betätigen der **Abbrechen**-Schaltfläche

Vorbereitung

Wir nutzen wieder eine Anwendung auf Basis der Vorlage **Visual Basic\Windows Desktop\WPF-App (.NET Framework)**. Dieser fügen wir ein neues Element des Typs **ADO.NET Entity Data Model** hinzu und legen für dieses den Namen **BestellverwaltungContext** fest. Dann fügen wir mit den Methoden der Datenbankdatei **Access-**

ZuWPFFormulare.accdb (siehe dortigen Readme-Bericht) ein Fenster auf Basis des Formulars **frmKundendetails** hinzu. Der Entwurf dieses Fensters sieht anschließend wie in Bild 3 aus.

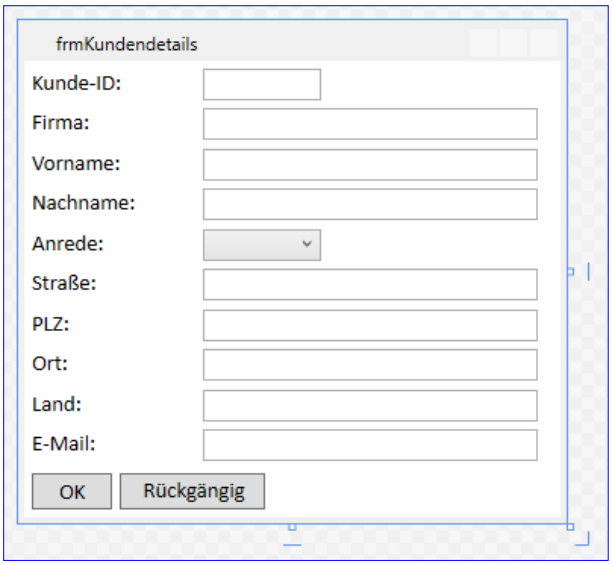


Bild 3: Entwurf des Fensters **frmKundendetails**

Nun wollen wir erreichen, dass die **Rückgängig**-Schaltfläche beim Anzeigen des Fensters deaktiviert ist und erst aktiviert wird, wenn der Benutzer bei mindestens einem Feld eine Änderung vorgenommen hat. Wenn der Benutzer die dann aktivierte Schaltfläche **Rückgängig** betätigt, sollen die Änderungen rückgängig gemacht werden beziehungsweise die Werte zum Zeitpunkt des Öffnens des Fensters wieder eingestellt werden.

Speichern beim Schließen

Wir beginnen mit den einfachen Teil: Die Schaltfläche **btnOK** soll den geänderten Datensatz in die Datenbank übertragen. Das erledigen wir mit der folgenden Ergänzung der Ereignismethode **btnOK_Click**:

```
Private Sub cmdOK_Click(sender As Object, e As RoutedEventArgs)
    dbContext.SaveChanges()
    Close()
End Sub
```

Ereignis beim Ändern auslösen

Dann wollen wir ein Ereignis finden, das durch das Ändern eines Eintrags und das Verlassen des Feldes ausgelöst wird. Das scheint uns ausreichend zu sein – wir müssen ja nicht nach der Eingabe eines jeden Buchstaben prüfen, ob sich etwas geändert hat. Der erste Kandidat für eine passende Ereignismethode heißt **LostFocus**. Also fügen wir das Attribut zum XAML-Code hinzu.

```
<TextBox x:Name="txtFirma" Text="{Binding Kunde.Firma}" Height="21" Margin="118,31,0,0" Width="221" LostFocus="txtFirma_
LostFocus"/>
```

Wenn wir **LostFocus="** eingeben und dann die Tabulator-Taste betätigen, wird automatisch der Methodenname **txtFirma_LostFocus** vorgeschlagen. Durch Betätigen von **F12** landen wir dann direkt bei dieser Methode im Code behind-Modul.

Mit der folgenden Testanweisung können wir gleich nach dem Starten der Anwendung testen, ob das Verlassen des Textfeldes **txtFirma** das Ereignis auslöst:

```
Private Sub txtFirma_LostFocus(sender As Object, e As RoutedEventArgs)
    MessageBox.Show("LostFocus")
End Sub
```

Herausfinden, ob eine Änderung vorliegt

Dieser Ereignismethode wollen wir nun Code hinzufügen, mit dem wir prüfen können, ob eine Änderung im aktuell angezeigten Element des Typs **Kunde** vorliegt.

Dazu benötigen wir zunächst einen Verweis auf den Namespace **System.Data.Entity.Infrastructure**, den wir im Kopf des Code behind-Moduls einfügen. **System.Data.Entity** benötigen wir später auch noch:

```
Imports System.Data.Entity.Infrastructure
Imports System.Data.Entity
```

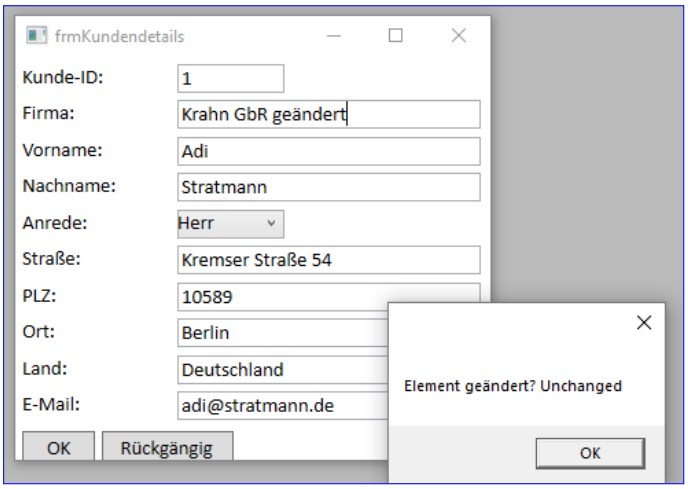


Bild 4: Aktueller Zustand des Elements

Danach erweitern wir die Ereignismethode wie folgt: Wir fügen eine Variable des Typs **DbEntityEntry** namens **GeaendertesElement** hinzu. Dieses füllen wir über die Auflistung **Entries** des **ChangeTracker**-Objekts, das dem aktuell in Kunde gespeicherten Element entspricht. Dieser Auflistung entnehmen wir mit der **First**-Eigenschaft das erste und einzige Element. Dann geben wir in einer weiteren **MessageBox** den aktuellen Zustand dieses Elements aus, den wir über die Eigenschaft **State** ermitteln:

```
Private Sub txtFirma_LostFocus(sender As Object, e As RoutedEventArgs)
    Dim GeaendertesElement As DbEntityEntry
    GeaendertesElement = dbContext.ChangeTracker.Entries().Where(Function(x) x.Entity Is Kunde).First()
    MessageBox.Show("Element geändert? " + GeaendertesElement.State.ToString())
End Sub
```

Das Ergebnis sieht wie in Bild 4 aus. Der Zustand **Unchanged** entspricht nun nicht dem, was wir erwartet haben.

Erst, wenn wir zum geänderten Datensatz zurückwechseln und dann erneut das Ereignis **LostFocus** auslösen, erscheint der Wert **Modified** im Meldungsfenster (siehe Bild 5).

Offensichtlich ist der neue Wert im zugrundeliegenden Element **Kunde** beim Auslösen des **LostFocus**-Ereignisses noch nicht angekommen.

Also ist **LostFocus** wohl nicht unser Ereignis. Dementsprechend kommen wir auch mit einem Ereignis wie **TextChanged** nicht weiter, denn möglicherweise benötigen wir einen abgeschlossenen Fokuswechsel, um die Änderung in das **Kunde**-Element zu übertragen.

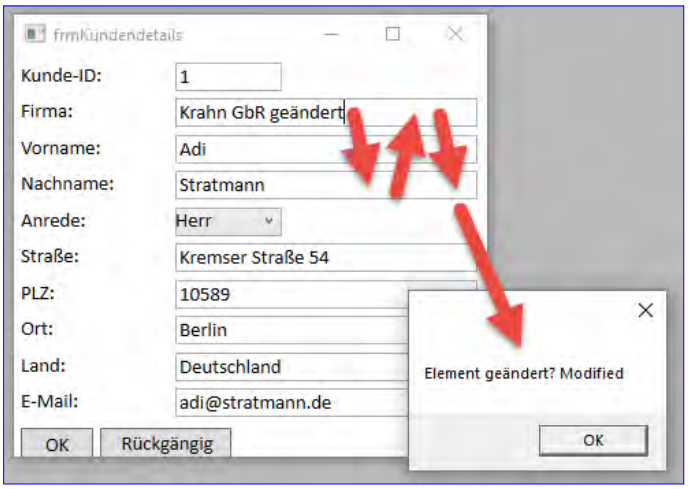


Bild 5: Aktueller Zustand des Elements, diesmal geändert

Änderung nach Fokuserhalt

Schauen wir uns also an, ob unsere Idee stimmt. Dazu fügen wir testweise dem nächsten Textfeld, hier **txtVorname**, das Attribut **GotFocus** hinzu:

```
<TextBox x:Name="txtVorname" Text="{Binding Kunde.Vorname}" Height="21" Margin="118,58,0,0" Width="221"
GotFocus="txtVorname_GotFocus"/>
```

Die Ereignismethode füllen wir wie folgt:

```
Private Sub txtVorname_GotFocus(sender As Object, e As RoutedEventArgs)
    Dim GeaendertesElement As DbEntityEntry
    GeaendertesElement = dbContext.ChangeTracker.Entries().Where(Function(x) x.Entity Is Kunde).First()
    MessageBox.Show("Element geändert? " + GeaendertesElement.State.ToString())
End Sub
```

Wenn wir nun den Inhalt von **txtFirma** ändern und den Fokus auf **txtVorname** verschieben, erhalten wir die erwartete Meldung – das Element hat den Zustand **Modified** (siehe Bild 6). Damit brauchen wir also für jedes Steuerelement, das den Fokus erhalten kann, das Ereignis **GotFocus** und eine entsprechende Ereignismethode.

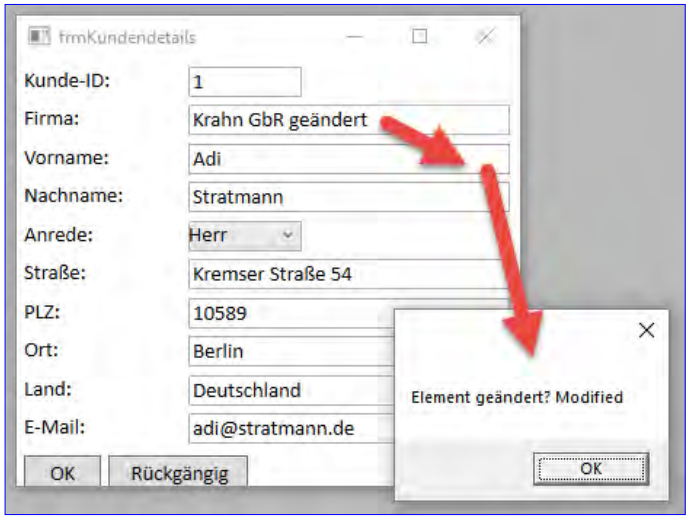


Bild 6: Beim Fokuserhalt ist das Element bereits geändert.

Das Zuweisen des Attributs zu allen Steuerelementen ist aufwendig – vor allem, je mehr Steuerelemente das Fenster enthält. Da das Ereignis immer die gleichen Anweisungen ausführen soll, reicht es allerdings aus, wenn wir nur ein Ereignis etwa namens **GotFocus** erstellen, dass wir dann für das Attribut **GotFocus** der verschiedenen Steuerelemente einrichten:

```
<TextBox x:Name="txtFirma" Text="{Binding Kunde.Firma}" Height="21" Margin="118,31,0,0" Width="221"
GotFocus="FrameworkElement_GotFocus"/>
<TextBox x:Name="txtVorname" Text="{Binding Kunde.Vorname}" Height="21" Margin="118,58,0,0" Width="221"
GotFocus="GotFocus"/>
...
```

Die Ereignismethode sähe dann so aus:

```
Private Sub FrameworkElement_GotFocus(sender As Object, e As RoutedEventArgs)
    ...
End Sub
```

Detailformulare mit Combo, Checkbox und Button

Im Artikel »Access zu WPF: Detailformulare mit Textfeldern« schauen wir uns an, wie die programmgesteuerten Möglichkeiten aussehen, um Formulare automatisch als WPF-Fenster oder -Seiten abzubilden. Damit haben wir einfache Detailformulare samt Textfeldern und Datenbindung unter WPF abgebildet. Nun wollen wir einen Schritt weitergehen und uns um weitere Steuerelemente wie etwa Kombinationsfelder und Kontrollkästchen kümmern. Außerdem wollen wir noch Schaltflächen zum Blättern in den Datensätzen sowie zum Anlegen neuer Datensätze hinzufügen.

Voraussetzungen

Wir gehen an dieser Stelle davon aus, dass Sie bereits ein Entity Data Model auf Basis des Access-Datenmodells erstellt und eine entsprechende SQL Server-Datenbank auf Basis des Entity Data Modells erstellt haben.

Wie das gelingt, zeigen die Artikel **Von Access zu Entity Framework: Datenmodell** und **Von Access zu Entity Framework: Daten** aus Ausgabe 5/2018. Die weiteren Vorarbeiten werden im Artikel **Access zu WPF: Detailformulare mit Textfeldern** erläutert.

Kombinationsfelder hinzufügen

Wenn wir etwa das Kombinationsfeld **cboAnredeID** im WPF-Fenster abbilden möchten, benötigen wir einige weitere Elemente. Das erste ist natürlich das **ComboBox**-Element im XAML-Code. Diesem müssen wir die Eigenschaften für die Bindung übergeben. Außerdem brauchen wir im Code behind-Modul der WPF-Seite zusätzlichen Code, der die Auflistung der Anreden als öffentliche Eigenschaft bereithält, damit wir vom XAML-Code aus auf die Anreden zugreifen können. Wir schauen uns als Erstes die notwendigen Änderungen an. Im XAML-Code fügen wir allgemeine Eigenschaften für den Steuerelementtyp **ComboBox** hinzu:

```
<Window x:Class="frmKundendetails" ... Title="frmKundendetails" Height="332" Width="362">
    <Window.Resources>
        ...
        <Style TargetType="{x:Type ComboBox}">
            <Setter Property="HorizontalAlignment" Value="Left"></Setter>
            <Setter Property="VerticalAlignment" Value="Top"></Setter>
            <Setter Property="FontFamily" Value="Calibri"></Setter>
            <Setter Property="FontSize" Value="11pt"></Setter>
        </Style>
    </Window.Resources>
```

Für das Kombinationsfeld selbst fügen wir das **ComboBox**-Element hinzu. Im Beispielformular, das die Daten eines Kunden anzeigen soll, wollen wir mit der folgenden Definition eines **ComboBox**-Elements die Daten der Tabelle **Anreden** zur Auswahl einfügen und die für den aktuellen Kunden ausgewählte Anrede anzeigen:

```
    <Grid>
        ...
        <Label x:Name="Bezeichnungsfeld4" Content="Anrede:" Padding="0" Height="21" Margin="5,111,0,0" Width="64"/>
        <ComboBox x:Name="MehrwertsteuersatzID" Padding="0" Height="21" Margin="118,111,0,0" Width="221"
                  ItemsSource="{Binding Anreden}" SelectedItem="{Binding Kunde.Anrede, ValidatesOnDataErrors=True}"
                  SelectedValuePath="ID" SelectionChanged="Anrede_SelectionChanged">
            <ComboBox.ItemTemplate>
                <DataTemplate>
                    <TextBlock>
                        <TextBlock.Text>
                            <MultiBinding StringFormat="{}{0}">
                                <Binding Path="Name" />
                            </MultiBinding>
                        </TextBlock.Text>
                    </TextBlock>
                </DataTemplate>
            </ComboBox.ItemTemplate>
        </ComboBox>
        ....
    </Grid>
</Window>
```

In der Code behind-Klasse fügen wir zunächst im allgemeinen Teil die private Variable für die Liste der Anreden hinzu sowie die öffentliche Eigenschaft, über welche diese dann verfügbar ist:

```
Private _Anreden As List(Of Anrede)

Public Property Anreden As List(Of Anrede)
    Get
        Return _Anreden
    End Get
    Set(value As List(Of Anrede))
        _Anreden = value
    End Set
End Property
```

Schließlich benötigen wir in den Konstruktor-Methoden jeweils eine Anweisung, welche die Auflistung der Anreden mit den Daten der **DbSet**-Auflistung **Anreden** füllt:

```
Public Sub New()
    ...
```

```
    Anreden = New List(Of Anrede)(dbContext.Anreden)
    DataContext = Me
End Sub

Public Sub New(lngID As Long)
    ...
    Anreden = New List(Of Anrede)(dbContext.Anreden)
    DataContext = Me
End Sub
```

Standardeigenschaften für ComboBox-Element hinzufügen

Das wären zunächst einmal die Anforderungen für die einzufügenden Elemente, die wir in den Prozeduren des Moduls **mdlAccessZuWPF** berücksichtigen müssen. Die Erweiterungen sind teilweise trivial wie etwa die für die Standardeigenschaften für die Kombinationsfelder beziehungsweise **ComboBox**-Elemente. Diese fügen wir der Prozedur **AttributeHinzufuegen** wie folgt hinzu:

```
Public Sub AttributeHinzufuegen(frm As Form, strXAML As String)
    strXAML = strXAML & "        <Window.Resources>" & vbCrLf
    ...
    With frm.DefaultControl(acComboBox)
        strXAML = strXAML & "        <Style TargetType=""{x:Type ComboBox}"">" & vbCrLf
        strXAML = strXAML & "            <Setter Property=""HorizontalAlignment"" Value=""Left""></Setter>" & vbCrLf
        strXAML = strXAML & "            <Setter Property=""VerticalAlignment"" Value=""Top""></Setter>" & vbCrLf
        strXAML = strXAML & "            <Setter Property=""FontFamily"" Value=""" & .FontName & """></Setter>" & vbCrLf
        strXAML = strXAML & "            <Setter Property=""FontSize"" Value=""" & .FontSize & "pt""></Setter>" & vbCrLf
        strXAML = strXAML & "        </Style>" & vbCrLf
    End With
    strXAML = strXAML & "        </Window.Resources>" & vbCrLf
End Sub
```

ComboBox-Element hinzufügen

Der Teil, der die XAML-Definition des **ComboBox**-Elements zusammenstellt, ist etwas umfangreicher als der für das Zusammenstellen eines **Label**- oder eines **TextBox**-Elements. Das liegt daran, dass wir ein paar mehr Informationen sammeln müssen – etwa, um herauszufinden, welche Tabelle beziehungsweise welches **DbSet**-Element die Daten für das aufgeklappte **ComboBox**-Element liefert. Genau genommen ist es sogar recht kompliziert – und umso komplizierter, je mehr mögliche Fälle man abdecken möchte.

Gehen wir einmal davon aus, dass wir etwa das Kombinationsfeld zur Auswahl der Anrede eines Kunden abbilden wollen. Dann hat dieses Kombinationsfeld beispielsweise die folgende Abfrage als Datensatzherkunft:

```
SELECT [tblAnreden].[AnredeID], [tblAnreden].[Anrede] FROM tblAnreden;
```

Daneben verwendet es die erste Spalte dieser Abfrage als gebundene Spalte und die zweite soll angezeigt werden. Schließlich ist das Kombinationsfeld an das Feld **AnredeID** der Tabelle **tblKunden** gebunden. Diese Informationen benötigen wir auch für das **ComboBox**-Element – genau genommen sogar einige davon abgeleitete Informationen. Wir wollen ja nicht den Namen der Tabelle **tblAnreden** für das Attribut **ItemsSource** angeben, sondern benötigen das **DbSet** des Entity Data Models, dass die Daten dieser Tabelle liefert.

Generiertes Entity Data Model nutzen

In den Artikeln **Von Access zu Entity Framework: Datenmodell** und **Von Access zu Entity Framework: Update 1** haben wir gezeigt, wie Sie aus dem Datenmodell einer Access-Datenbank ein Entity Data Model erstellen können. Dort haben wir unter anderem eine Collection zusammengestellt, welche die resultierenden Mappings enthält – also etwa die Namen der Tabellen (zum Beispiel **tblKunden**), der resultierenden Klasse (**Kunde**), des **DbSet**-Elements (**Kunden**) und weiterer Informationen.

Die Daten aus dieser Collection können wir auch noch nutzen, um die Informationen zur Definition der **ComboBox**-Elemente zusammenzustellen. In der Collection, die Elemente des Typs **clsMapping** aufnimmt, fehlen allerdings noch zwei Informationen, die wir noch benötigen: Wenn der Name eines Feldes mit dem resultierenden Namen der Klasse übereinstimmt, was etwa bei der Tabelle **tblAnreden** mit den Feldern **AnredeID** und **Anrede** auftritt, dann soll der Name dieses Feldes als Eigenschaft der Klasse in **Name** umbenannt werden. Da wir zufällig genau beim Beispiel der Tabelle **tblKunden**, welche ja im Fremdschlüsselfeld **AnredeID** auf die Datensätze der Tabelle **tblAnreden** zugreift, auf diesen Fall treffen, wissen wir auch, dass diese Informationen gut noch in den Elementen der Collection **colMappings** untergebracht werden könnten.

Dazu erweitern wir zunächst die Klasse **clsMapping** in der Access-Datenbank, welche die zu migrierenden Formulare enthält, um die folgenden beiden privaten Variablen und die entsprechenden Getter und Setter:

```
Private m_RenamedField As String
Private m_RenamedFieldOriginal As String

Public Property Get RenamedField() As String
    RenamedField = m_RenamedField
End Property

Public Property Let RenamedField(str As String)
    m_RenamedField = str
End Property

Public Property Get RenamedFieldOriginal() As String
    RenamedFieldOriginal = m_RenamedFieldOriginal
End Property

Public Property Let RenamedFieldOriginal(str As String)
    m_RenamedFieldOriginal = str
End Property
```

Die Eigenschaft **RenamedFieldOriginal** soll dann etwa den Namen des Feldes **Anrede** aufnehmen und **RenamedField** den Namen der resultierenden Eigenschaft der Klasse **Anrede**, also **Name**. Die Prozedur **EDMErstellen** aus dem Modul **mdlEDM** wandeln wir in eine gleichnamige Funktion um, welche eine Collection als Funktionswert zurückliefern soll:

```
Public Function EDMErstellen() As Collection
    ...
    Set colMappings = New Collection
    ...
```

In der **For Each**-Schleife über alle Mappings, in denen die Funktion die Felder untersucht, schreibt die Funktion dann die entsprechenden Werte in die beiden Eigenschaften **RenamedField** und **RenamedFieldOriginal** der **clsMapping**-Klasse. Das sieht dann wie folgt aus:

```
    For Each objMapping In colMappings
        With objMapping
            ...
            For Each fld In tdf.Fields
                ...
                If fld.Name = .Entity_Original Then
                    strFieldname = "Name"
                    objMapping.RenamedField = "Name"
                    objMapping.RenamedFieldOriginal = fld.Name
                Else
                    ...
                End If
                ...
            Next fld
            ...
        End With
        ...
    Next objMapping
    ...
```

Schließlich wird die Collection **colMappings** als Funktionswert der Funktion **EDMErstellen** zurückgegeben:

```
    Set EDMErstellen = colMappings
End Function
```

Anpassung der Prozeduren zum Migrieren des Formulars

Die Funktion **EDMErstellen** rufen wir nun nicht mehr nur auf, wenn wir das Entity Data Model auf Basis des Datenmodells der Datenbank erstellen wollen, sondern auch beim Migrieren eines der Formulare mit der Prozedur **FormularNachWPF** des

Moduls **mdlAccessZuWPF** der Access-Datenbank. Dies erledigen wir direkt in der ersten Anweisung nach dem Deklarationsteil der Prozedur:

```
Public Sub FormularNachWPF(strForm As String, strKlasse As String)
    ...
    Dim colMappings As Collection
    Set colMappings = EDMErstellen
    ...
```

Dann übergeben wir diese Collection beim Aufruf der Prozedur **SteuerelementeHinzufuegen** als neuen Parameter:

```
    SteuerelementeHinzufuegen frm, strXAML, strKlasse, colMappings
    ...
End Sub
```

Die Prozedur **SteuerelementeHinzufuegen** erfährt die umfassendsten Anpassungen. Zunächst fügen wir **colMappings** mit dem Datentyp **Collection** als vierten Parameter zur Prozedur hinzu:

```
Public Function SteuerelementeHinzufuegen(frm As Form, strXAML As String, strKlasse As String, colMappings As Collection)
    ...
```

Dann benötigen wir die folgenden zusätzlichen Variablen. Die erste namens **strEntitiesCombo** nimmt den Namen der **DbSet**-Auflistung für die Elemente des **ComboBox**-Elements auf (etwa **Anreden**), das zweite namens **strEntityCombo** den Namen der Klasse für die Auflistung. **strPKCombo** erhält das Primärschlüsselfeld der entsprechenden Entität, also zum Beispiel **ID**.

strSichtbaresFeldCombo schließlich füllen wir mit dem Namen des zweiten Feldes der Datensatzherkunft des Kombinationsfeldes, im Falle der Datensatzherkunft **SELECT [tblAnreden].[AnredeID], [tblAnreden].[Anrede] FROM tblAnreden;** also etwa das Feld **Anrede**.

Diesen Wert ändern wir aber in diesem Fall noch, da **Anrede** ja dem Namen der entstehenden Entität entspricht und dementsprechend umbenannt werden muss.

```
    Dim strEntitiesCombo As String
    Dim strEntityCombo As String
    Dim strPKCombo As String
    Dim strSichtbaresFeldCombo As String
    ...
```

In der Schleife über alle Steuerelemente des Formulars landen wir dann früher oder später bei einem Kombinationsfeld. Für dieses haben wir in einer **Select Case**-Bedingung einen eigenen Zweig eingerichtet. In diesem fügen wir zunächst die Schriftart und Schriftgröße ein, sollte sich diese von der Standardschriftart für dieses Steuerelement unterscheiden:

```
For Each ctl In frm.Controls
    ...
    Select Case ctl.ControlType
        ...
        Case acComboBox
            If Not ctl.FontName = strFontFamilyTextBox Then
                 strFontname = "FontFamily=""" & strFontFamilyTextBox & """ "
            End If
            If Not ctl.FontSize = strFontSizeTextBox Then
                strFontSize = "FontSize=""" & strFontSizeTextBox & "pt"" "
            End If
```

Danach füllen wir die Variable **strFeldname** mit dem Wert des Feldes **Steuerelementinhalt** des Kombinationsfeldes:

```
            strFeldname = ctl.ControlSource
```

Die Variable **strSichtbaresFeldCombo** nimmt das zweite Feld der als Datensatzherkunft angegebenen Tabelle oder Abfrage als Wert:

```
            strSichtbaresFeldCombo = CurrentDb.OpenRecordset(ctl.RowSource).Fields(1).Name
```

In einer **If...Then**-Bedingung verwenden wir dann das Ergebnis eines Aufrufs der Funktion **GetDbSetCombo** als Kriterium. Liefert diese Funktion, welche aus den **Mapping**-Informationen aus **colMappings** und der Datensatzherkunft des Steuerelements den Namen der Entitätenliste, des Primärschlüsselfeldes und des sichtbaren Feldes ermitteln soll, den Wert **True**, wurden diese Werte erfolgreich gefüllt und wir können den Code für das **ComboBox**-Element zusammenstellen:

```
            If GetDbSetCombo(colMappings, ctl.RowSource, strEntitiesCombo, strPKCombo, strSichtbaresFeldCombo, _
                strEntityCombo) Then
```

Ist das der Fall, tragen wir zunächst den Namen des Steuerelements für das Attribut **Name** des **ComboBox**-Elements ein sowie die Höhe, die Breite und die Abstände zum linken und oberen Rand:

```
                strXAML = strXAML & "<ComboBox x:Name=""" & ctl.Name & """ Padding=""0"" Height=""" & lngHeight _
                    & """ Margin=""" & lngLeft & "," & lngTop & ",0,0"" Width=""" & lngWidth & """" & vbCrLf
```

Für das Attribut **ItemsSource** stellen wir eine Bindung zu dem Element aus **strEntitiesCombo** her, in diesem Fall **Anreden**:

```
                strXAML = strXAML & "            ItemsSource = ""{Binding " & strEntitiesCombo & "}""" & vbCrLf
```

Den aktuell ausgewählten Eintrag wollen wir über das Feld **Name** der Klasse **Anrede** ermitteln und tragen diesen für das Attribut **SelectedItem** ein:

```
strXAML = strXAML & "            SelectedItem = ""{Binding " & strKlasse & "." & strEntityCombo _
    & ", ValidatesOnDataErrors=True}""" & vbCrLf
```

Zu den allgemeinen Attributen gehört auch noch **SelectedValuePath**, das festlegt, welche Eigenschaft der gebundenen Spalte entspricht, also wonach der Eintrag der **ComboBox** für den aktuellen Datensatz im Fenster ausgewählt wird:

```
strXAML = strXAML & "            SelectedValuePath=""" & strPKCombo & """> " & vbCrLf
```

Danach folgen die anzuzeigenden Werte des **ComboBox**-Elements. Dazu legen wir das folgende Konstrukt an, mit dem Sie direkt ein **MultiBinding**-Element haben, das Sie nach der Migration leicht um weitere Felder erweitern können. Wir fügen hier für das Element **Binding** die Eigenschaft auf **strSichtbaresFeldCombo** ein, was im Beispiel der Bezeichnung **Name** entspricht:

```
strXAML = strXAML & "    <ComboBox.ItemTemplate>" & vbCrLf
strXAML = strXAML & "        <DataTemplate>" & vbCrLf
strXAML = strXAML & "            <TextBlock>" & vbCrLf
strXAML = strXAML & "                <TextBlock.Text>" & vbCrLf
strXAML = strXAML & "                    <MultiBinding StringFormat=""{}{0}"">" & vbCrLf
strXAML = strXAML & "                        <Binding Path=""" & strSichtbaresFeldCombo & """ />" & vbCrLf
strXAML = strXAML & "                    </MultiBinding>" & vbCrLf
strXAML = strXAML & "                </TextBlock.Text>" & vbCrLf
strXAML = strXAML & "            </TextBlock>" & vbCrLf
strXAML = strXAML & "        </DataTemplate>" & vbCrLf
strXAML = strXAML & "    </ComboBox.ItemTemplate>" & vbCrLf
strXAML = strXAML & "</ComboBox>" & vbCrLf
```

Wenn **GetDbSetCombo** den Wert **False** liefert, konnte kein zur Datensatzherkunft des Kombinationsfeldes passendes Mapping gefunden werden. Das ist etwa der Fall, wenn als Datensatzherkunft eine Abfrage verwendet wurde. Dann erscheint eine entsprechende Meldung:

```
            Else
                MsgBox "ComboBox für das Steuerelement '" & ctl.Name & "' konnte nicht erstellt werden, da die " _
                    & "RowSource keine passende Tabelle enthielt."
            End If
        ...
    End Select
Next ctl
End Function
```

Nun schauen wir uns noch die Funktion **GetDbSetCombo** an. Dieser übergeben wir die Collection mit den Mapping-Informationen in den verschiedenen **clsMapping**-Klassen sowie die Datensatzherkunft des Kombinationsfelds als Parameter. Die übrigen Parameter sollen in der Funktion gefüllt beziehungsweise angepasst werden:

Access zu WPF: Validierung und Navigation

Im Artikel »Access zu WPF: Detailformulare mit Combo, Checkbox und Button« haben wir gezeigt, wie Sie die in Formularen enthaltenen gebundenen Steuerelemente wie Kombinationsfelder und Kontrollkästchen sowie Schaltflächen in WPF-Fenster übertragen. In diesem Artikel wollen wir uns ansehen, wie Sie einem WPF-Fenster auf Basis eines Access-Detailformulars eine Validierung und Navigationsschaltflächen hinzufügen. Die notwendigen Techniken haben wir bereits im Artikel »EDM: Blättern in Datensätzen« vorgestellt, nun integrieren wir diese in unsere VBA-Prozedur zum Erstellen des Codes für ein WPF-Fenster mit Navigationsschaltflächen.

Um festzulegen, ob ein Formular mit oder ohne Validierung und Navigationsschaltflächen erstellt werden soll, fügen wir der Prozedur **FormularNachWPF** zwei weitere Parameter hinzu:

- **bolValidation**: Gibt an, ob die Validierung hinzugefügt werden soll.
- **bolNavigation**: Gibt an, ob die Navigationsschaltflächen hinzugefügt werden sollen.

Die Validierung für einen neuen, leeren Datensatz sieht dann etwa wie in Bild 1 aus.

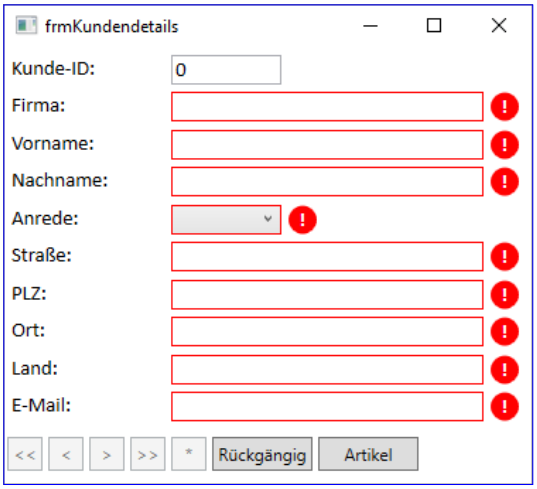


Bild 1: Validierung bei einem neuen, leeren Datensatz

Die Prozedur **FormularNachWPF** erweitern wir an den folgenden Stellen um die Entgegennahme beziehungsweise Übergabe der Parameter:

```
Public Sub FormularNachWPF(strForm As String, strTabelle As String, Optional bolValidation As Boolean, _
        Optional bolNavigation As Boolean)
    ...
    lngHeight = TwipsToPixelsHeight(frm.Section(0).Height) + 30
    If bolNavigation = True Then
        lngTopNavi = lngHeight - 30
        lngHeight = lngHeight + 28
    End If
```

Bei der Definition des **Window**-Elements fügen wir noch das Ereignisattribut **Closing** mit dem Wert **Window_Closing** hinzu, welches sicherstellt, dass das Fenster nur bei geändertem und gespeichertem Datensatz geschlossen werden kann:

```
    strXAML = strXAML & "<Window x:Class=""" & frm.Name & """" & vbCrLf
```

```
    ...
    strXAML = strXAML & "        Title=""" & strTitle & """ Height=""" & lngHeight & """ Width=""" & lngWidth _
        & """ Closing=""Window_Closing"">" & vbCrLf
```

Für die folgenden Aufruf haben wir die Übergabe der Parameter **bolValidation** und **bolNavigation** erweitert:

```
    ...
    AttributeHinzufuegen frm, strXAML, bolValidation, bolNavigation
    ...
    SteuerelementeHinzufuegen frm, strXAML, strTabelle, colMappings, bolValidation, bolNavigation
    If bolNavigation Then
        NavigationHinzufuegen strXAML, lngTopNavi
    End If
    ...
End Sub
```

Erweiterung der Prozedur AttributeHinzufuegen

Die XAML-Datei enthält mit eingebauter Validierung einige Zeilen Code mehr. Diesem werden wir in der Prozedur **AttributeHinzufuegen** wie folgt gerecht, die ebenfalls die beiden Parameter **bolValidation** und **bolNavigation** erwartet:

```
Public Sub AttributeHinzufuegen(frm As Form, strXAML As String, bolValidation As Boolean, bolNavigation As Boolean)
```

In dieser Prozedur geht es allein um das Zusammenstellen allgemeiner Properties für Steuerelemente im **WindowResources**-Bereich. Da wir hier auch die Steuerelemente durchlaufen, um die **DataTrigger** und **MultiDataTrigger** für die Validierung zu definieren, benötigen wir eine Variable namens **ctl** mit dem Typ **Control**:

```
    Dim ctl As Control
```

Danach geht es dann direkt rund. Wir prüfen, ob **bolValidation** den Wert **True** aufweist und fügen dann die Vorlage für die Markierung nicht erfolgreich validierter Steuerelemente hinzu:

```
    strXAML = strXAML & "        <Window.Resources>" & vbCrLf
    If bolValidation Then
        strXAML = strXAML & "        <Style TargetType=""{x:Type FrameworkElement}"">" & vbCrLf
        strXAML = strXAML & "            <Setter Property=""Validation.ErrorTemplate"">" & vbCrLf
        strXAML = strXAML & "                <Setter.Value>" & vbCrLf
        strXAML = strXAML & "                    <ControlTemplate>" & vbCrLf
        strXAML = strXAML & "                        <DockPanel>" & vbCrLf
        strXAML = strXAML & "                            <Border Background=""Red"" DockPanel.Dock=""right"" " _
            & "Margin=""5,0,0,0"" Width=""20"" Height=""20"" CornerRadius=""10""" & vbCrLf
        strXAML = strXAML & "                                ToolTip=""{Binding ElementName=customAdorner, " _
```

```
        & "Path=AdornedElement.(Validation.Errors)[0].ErrorContent}"">" & vbCrLf
    strXAML = strXAML & "                                <TextBlock Text=""!"" VerticalAlignment=""center"" " _
        & "HorizontalAlignment=""center""" & vbCrLf
    strXAML = strXAML & "                                    FontWeight=""Bold"" Foreground=""white"" />" & vbCrLf
    strXAML = strXAML & "                            </Border>" & vbCrLf
    strXAML = strXAML & "                            <AdornedElementPlaceholder Name=""customAdorner"" " _
        & "VerticalAlignment=""Center"" >" & vbCrLf
    strXAML = strXAML & "                                <Border BorderBrush=""red"" BorderThickness=""1"" />" & vbCrLf
    strXAML = strXAML & "                            </AdornedElementPlaceholder>" & vbCrLf
    strXAML = strXAML & "                        </DockPanel>" & vbCrLf
    strXAML = strXAML & "                    </ControlTemplate>" & vbCrLf
    strXAML = strXAML & "                </Setter.Value>" & vbCrLf
    strXAML = strXAML & "            </Setter>" & vbCrLf
    strXAML = strXAML & "        </Style>" & vbCrLf
End If
...
```

Der nächste Teil, der nur von der Notwendigkeit einer Validierung abhängt, sind die Elemente **ComboBox** und **TextBox**, die wir beide mit den **Style**-Attributen für die Validierung ausstatten müssen – also mit dem soeben angelegten Style. Dazu prüfen wir wieder per **If...Then**-Bedingung, ob **bolValidation** den Wert **True** aufweist, und fügen dann eine der beiden Zeilen als erste Zeile des Styles hinzu:

```
...
With frm.DefaultControl(acTextBox)
    If bolValidation Then
        strXAML = strXAML & "        <Style TargetType=""{x:Type TextBox}"" "_
            & "BasedOn=""{StaticResource {x:Type FrameworkElement}}"">" & vbCrLf
    Else
        strXAML = strXAML & "        <Style TargetType=""{x:Type TextBox}"">" & vbCrLf
    End If
    ...
End With
With frm.DefaultControl(acComboBox)
    If bolValidation Then
        strXAML = strXAML & "        <Style TargetType=""{x:Type ComboBox}"" " _
            & "BasedOn=""{StaticResource {x:Type FrameworkElement}}"">" & vbCrLf
    Else
        strXAML = strXAML & "        <Style TargetType=""{x:Type ComboBox}"">" & vbCrLf
    End If
    ...
End With
```

```
...
```

Mit dem **BasedOn**-Attribut sorgen wir dafür, dass der oben definierte, für den Typ **FrameworkElement** vorgesehene Style sowohl an **ComboBox**- als auch **TextBox**-Elemente vererbt wird. Durch das Erfassen der Attribute für die Markierung nicht erfolgreich validierter Steuerelemente in einem eigenen **Style**-Element sparen wir einige Zeilen Code, da wir die gleichen Elemente nicht für beide Steuerelementtypen einzeln hinterlegen müssen.

Erst wollten wir auch das **CheckBox**-Element auf diese Weise ausstatten, aber dieses ist ja ohnehin meist mit dem Wert **False** vorbelegt, sodass keine Validierung notwendig ist.

Danach stellen wir die **DataTrigger**-Elemente für drei Steuerelementtypen **ComboBox**, **TextBox** und **CheckBox** zusammen, die dafür sorgen, dass die Eigenschaft **IsEnabled**, die wir später im Code behind-Modul definieren, für die mit diesem **DataTrigger** ausgestatteten Style auf **False** eingestellt wird, wenn die Validierung auch nur für ein einziges der Steuerelemente fehlschlägt. Dies betrifft etwa die Schaltflächen zum Navigieren, die wir später hinzufügen. Hier verwenden wir zum ersten Mal die Variable **ctl**, mit der wir in einer **For Each**-Schleife alle Steuerelemente des Formulars durchlaufen. Damit fügen wir dem **Style**-Element, das wir über das Attribut **x:Key** mit dem Wert **EnableOnValidation** identifizieren, für jedes Steuerelement drei Zeilen hinzu, die das **DataTrigger**-Element mit der Bedingung und das **Setter**-Element mit dem zu setzenden Wert enthalten:

```
    ...
    If bolValidation Then
        strXAML = strXAML & "        <Style x:Key=""EnableOnValidation"" TargetType=""{x:Type Button}"">" & vbCrLf
        strXAML = strXAML & "            <Style.Triggers>" & vbCrLf
        For Each ctl In frm.Controls
            Select Case ctl.ControlType
                Case acComboBox, acCheckBox, acTextBox
                    If Not Len(ctl.ControlSource) = 0 Then
                        strXAML = strXAML & "                <DataTrigger Binding=""{Binding ElementName=" & ctl.Name _
                            & ", Path=(Validation.HasError)}"" Value=""True"">" & vbCrLf
                        strXAML = strXAML & "                    <Setter Property=""IsEnabled"" " _
                            & "Value=""False""></Setter>" & vbCrLf
                        strXAML = strXAML & "                </DataTrigger>" & vbCrLf
                    End If
            End Select
        Next ctl
        strXAML = strXAML & "            </Style.Triggers>" & vbCrLf
        strXAML = strXAML & "        </Style>" & vbCrLf
...
```

Außerdem benötigen wir noch jeweils ein **MultiDataTrigger**-Element für jedes gebundene Steuerelement der Typen **ComboBox**, **TextBox** und **CheckBox**. Dieses liefert standardmäßig für das Attribut **IsEnabled** den Wert **True**. Wenn alle in den **Condition**-Elementen genannten Bedingungen wahr sind, also die Validierung für alle Elemente keinen Fehler liefert, wird **IsEnabled**

auf **False** eingestellt. Dieser mit dem Schlüssel **EnableOnFailedValidation** benannte Style wird für die **Rückgängig**-Schaltfläche eingesetzt, um diese nur zu aktivieren, wenn mindestens eine Validierung fehlschlägt und diese nur zu deaktivieren, wenn alle Validierungen erfolgreich waren. Auch hier durchlaufen wir in einer **For Each**-Schleife alle gebundenen Steuerelemente:

```
...
        strXAML = strXAML & "        <Style x:Key=""EnableOnFailedValidation"" TargetType=""{x:Type Button}"">" & vbCrLf
        strXAML = strXAML & "            <Setter Property=""IsEnabled"" Value=""True"" />" & vbCrLf
        strXAML = strXAML & "            <Style.Triggers>" & vbCrLf
        strXAML = strXAML & "                <MultiDataTrigger>" & vbCrLf
        strXAML = strXAML & "                    <MultiDataTrigger.Conditions>" & vbCrLf
        For Each ctl In frm.Controls
            Select Case ctl.ControlType
                Case acComboBox, acCheckBox, acTextBox
                    If Not Len(ctl.ControlSource) = 0 Then
                        strXAML = strXAML & "                        <Condition Binding=""{Binding ElementName=" _
                            & ctl.Name & ", Path=(Validation.HasError)}"" Value=""False"" />" & vbCrLf
                    End If
            End Select
        Next ctl
        strXAML = strXAML & "                    </MultiDataTrigger.Conditions>" & vbCrLf
        strXAML = strXAML & "                    <Setter Property=""IsEnabled"" Value=""False"" />" & vbCrLf
        strXAML = strXAML & "                </MultiDataTrigger>" & vbCrLf
        strXAML = strXAML & "            </Style.Triggers>" & vbCrLf
        strXAML = strXAML & "        </Style>" & vbCrLf
    End If
```

Sollen die Navigationsschaltflächen angezeigt werden, brauchen wir auch noch diese beiden **Style**-Elemente, die dafür sorgen, dass die Schaltfläche **btnErster** und **btnVorheriger**, die wir nachfolgend hinzufügen, nur aktiviert werden, wenn gerade nicht

AUSGABE 01/2020

DATENBANK ENTWICKLER

MAGAZIN FÜR DIE DATENBANKENTWICKLUNG MIT VISUAL STUDIO FÜR DESKTOP, WEB UND CO.

TOP-THEMEN:

WPF	FlowDocument-Elemente mit XAML	SEITE 3
WPF-CONTROLS	Das RichTextBox-Steuerelement	SEITE 15
WPF-CONTROLS	Format-Menü für die RichTextBox	SEITE 26
VISUAL STUDIO	Codeschnipsel in Visual Studio	SEITE 39
LÖSUNGEN	Onlinebanking mit DDBAC II: Überweisungen	SEITE 54

André Minhorst Verlag

FlowDocument-Elemente mit XAML

Ein Element des Typs FlowDocument dient zusammen mit den drei Steuerelementen FlowDocumentScrollViewer, FlowDocumentTextViewer und FlowDocumentReader der Textdarstellung in WPF. Während auch andere Steuerelemente wie TextBox oder TextBlock Texte darstellen können und auch ihre Bearbeitung ermöglichen, bietet das FlowDocument viel mehr Möglichkeiten, was die Gestaltung des Texts angeht. Neben verschiedenen Absatzformatierungen und Zeichenformatierungen sind hier auch die Anzeige von Listen, Tabellen oder eingebauten Elementen wie Bildern möglich. Dieser Artikel stellt das FlowDocument-Element mit seinen Eigenschaften vor und zeigt auch, wie Sie solche Dokumente mit VB zur Laufzeit erstellen können. Letzteres ist interessant, wenn Sie Inhalte aus verschiedenen Feldern einer Datenbank strukturiert und formatiert darstellen wollen.

FlowDocuments benötigen entweder eines der drei Darstellungs-Steuerelemente **FlowDocumentScrollViewer**, **FlowDocumentTextViewer** oder **FlowDocumentReader** oder, wenn Sie den Inhalt bearbeiten wollen, das **RichTextBox**-Steuerelement. Im vorliegenden Artikel verwenden wir einfach das **FlowDocumentReader**-Steuerelement, damit wir die per WPF zusammengestellten formatierten Texte darstellen können. Auf das **RichTextBox**-Steuerelement gehen wir im Artikel **Das RichTextBox-Steuerelement** ein.

FlowDocument anlegen

Ein **FlowDocument**-Element können wir beispielsweise wie in Bild 1 in einem **FlowDocumentReader**-Steuerelement ausgeben. Und die WPF-Definition beginnt auch genau so: Wir legen in einem neuen, leeren Fenster ein **FlowDocumentReader**-Element an, dem wir ein **FlowDocument**-Element hinzufügen:

```
<FlowDocumentReader>
    <FlowDocument>
    ...
    </FlowDocument>
```

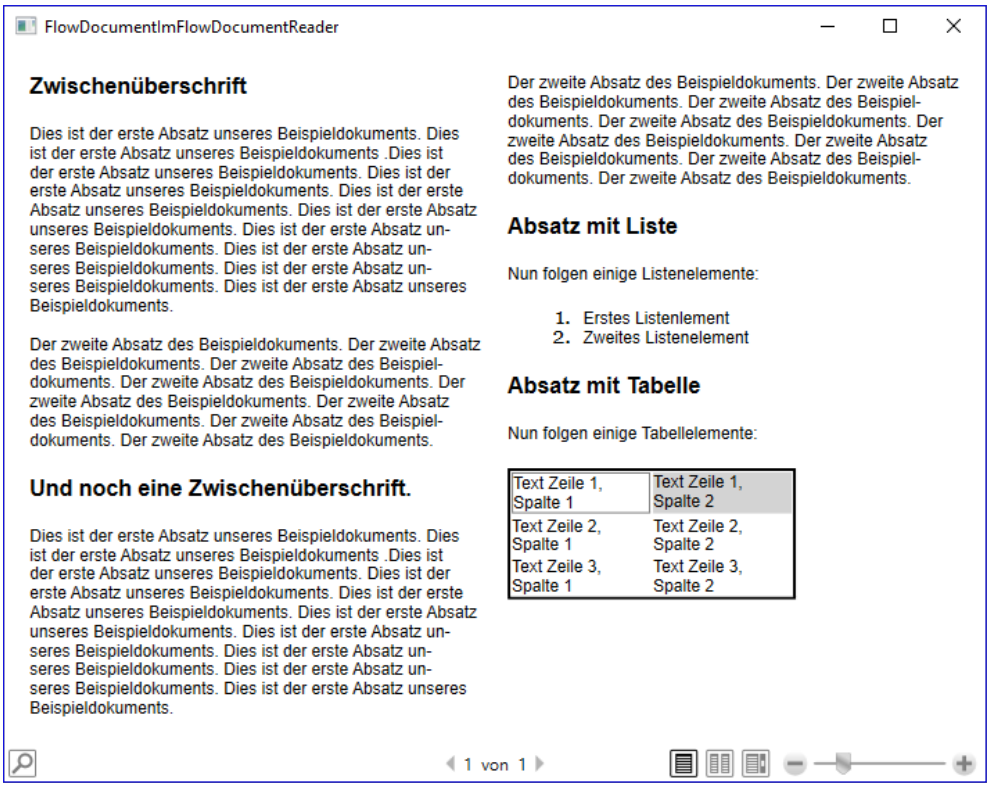


Bild 1: Ausgabe des Inhalts eines FlowDocuments im **FlowDocumentReader**-Steuerelement

```
</FlowDocumentReader>
```

Damit erhalten wir dann ein noch leeres **FlowDocumentReader**-Element im Entwurf der **.xaml**-Datei (siehe Bild 2).

Bild 2: Das leere **FlowDocument** im **FlowDocumentReader**

Text zum FlowDocument hinzufügen

Das FlowDocument füllen wir sehr schnell, indem wir einfach ein **Paragraph**-Objekt mit einen beliebigen Text zum **FlowDocument**-Element hinzufügen:

```
<Paragraph>Ein neuer Text.</Paragraph>
```

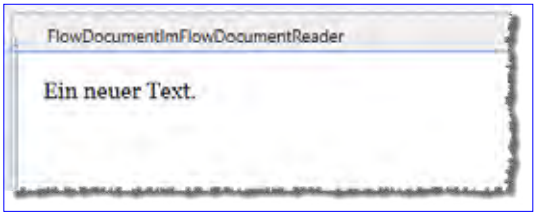


Bild 3: Ein Beispielabsatz

Praktischerweise zeigt der Entwurf die so hinzugefügten Inhalte direkt an (siehe Bild 3). So müssen wir nicht immer das Projekt starten, um den Fortschritt zu betrachten.

Absatzformatierungen

Das **Paragraph**-Element erlaubt einige Formatierungen, die für den kompletten Absatz gelten. Diese geben Sie als Attribute für das **Paragraph**-Element an. Die wichtigsten dienen der Formatierung des Texts:

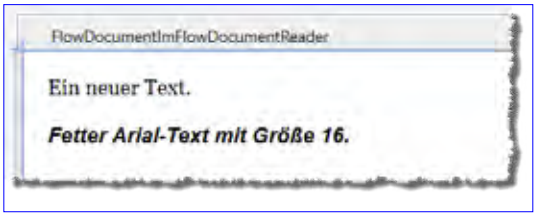


Bild 4: Formatierter Absatz

- **FontFamily**: Gibt den Namen der Schriftart für den Text des Absatzes an, zum Beispiel **Arial**.
- **FontSize**: Gibt an, welche Größe die Schrift haben soll (Angabe als numerischer Wert wie etwa in Word)
- **FontWeight**: Gibt an, ob der Text normal (Standardwert) oder fett gedruckt werden soll (Wert **Bold**)
- **FontStyle**: Für kursive Darstellung mit den Werten **Italic** oder **Oblique**.

Mit diesen Formatierungen erhalten wir bereits ein anderes Ergebnis als zuvor (siehe Bild 4):

```
<Paragraph FontWeight="Bold" FontFamily="Arial" FontSize="16" FontStyle="Italic">Fetter Arial-Text mit Größe 16.</Para-
graph>
```

Damit können wir verschiedene Absatzformatierungen durchführen und erhalten eine optisch brauchbare Formatierung – wie zum Beispiel mit den wie folgt definierten Absätzen:

```
<Paragraph FontSize="16" FontFamily="Arial" FontWeight="Bold">Zwischenüberschrift</Paragraph>
<Paragraph FontSize="12" FontFamily="Arial">Dies ist der erste Absatz unseres Beispieldokuments. ...</Paragraph>
<Paragraph FontSize="12" FontFamily="Arial">Der zweite Absatz des Beispieldokuments. ...</Paragraph>
<Paragraph FontSize="16" FontFamily="Arial" FontWeight="Bold">Und noch eine Zwischenüberschrift.</Paragraph>
<Paragraph FontSize="12" FontFamily="Arial">Dies ist der erste Absatz unseres Beispieldokuments. ...</Paragraph>
<Paragraph FontSize="12" FontFamily="Arial">Der zweite Absatz des Beispieldokuments. ...</Paragraph>
```

Diese Absätze sehen nach dem Starten des Projekts wie in Bild 5 aus. Wir haben das Fenster mit dem **FlowDocumentReader** ein wenig verkleinert, um zu zeigen, dass das Dokument dann auf zwei Seiten aufgeteilt wird, von der die erste angezeigt wird. Mit den Steuerelementen unten können Sie dann zur zweiten Seite blättern. Außerdem haben wir die Breite soweit verringert, dass das Fenster den Text in nur einer Spalte anzeigt. Wenn Sie das Fenster wieder breiter ziehen, erscheinen zunächst zwei Spalten, dann auch noch mehr als zwei Spalten.

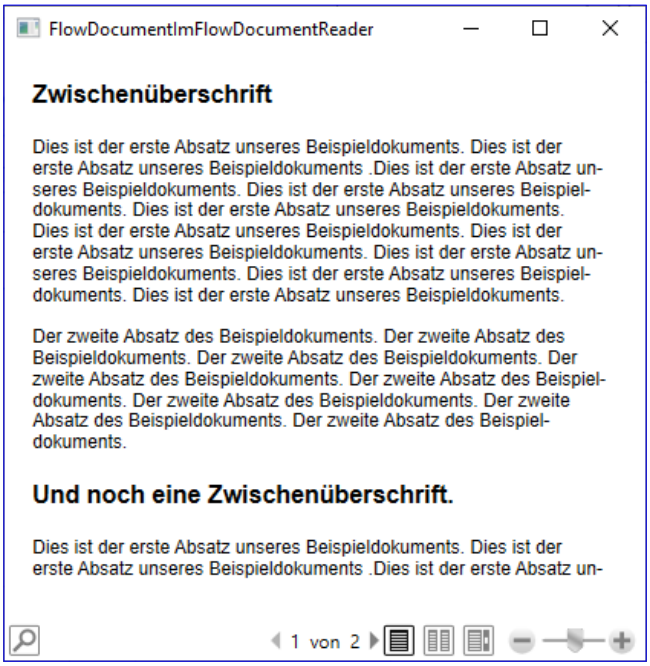


Bild 5: Ausgabe einiger Absätze im **FlowDocumentReader**-Element

Die Formatierung der Schrift müssen Sie nicht für jeden Absatz einzeln durchführen. Sie können die Attribute auch für die übergeordneten Elemente definieren. Die untergeordneten Elemente übernehmen beziehungsweise »erben« die Werte der übergeordneten Elemente.

Inline-Elemente

Sie können auch einzelne Elemente innerhalb eines Absatzes formatieren. Dazu nutzen Sie wie unter HTML sogenannte Inline-Formatierungen. Zum Formatieren des Textes mit fetten, kursiven oder unterstrichenen Zeichen verwenden Sie die folgenden Elemente:

- **Bold**: Formatiert den eingeschlossenen Text fett.
- **Italic**: Formatiert den eingeschlossenen Text kursiv.
- **Underline**: Unterstreicht den eingeschlossenen Text.

Ein Beispiel mit allen drei Formatierungen:

```
<Paragraph FontSize="12" FontFamily="Arial">Text mit <Bold>fettem</Bold>, <Italic>kursivem</Italic> und
<Underline>unterstrichenem</Underline> Inhalt.</Paragraph>
```

In den **.xaml**-Code eingebaute Zeilenumbrüche wirken sich übrigens nicht auf die Darstellung aus. Die folgende, vom **.xaml**-Editor automatischen hinzugefügten Zeilenumbrüche werden also nicht auf den Text übertragen:

```
<Paragraph FontSize="12" FontFamily="Arial">Text mit
    <Bold>fettem</Bold> ,
    <Italic>kursivem</Italic> und
    <Underline>unterstrichenem</Underline> Inhalt.
</Paragraph>
```

Zeilenumbrüche

Wenn Sie Zeilenumbrüche wünschen, müssen Sie diese explizit angeben. Unter HTML fügen Sie Zeilenumbrüche mit dem Zeichen **
** ein, unter WPF in **FlowDocument**-Elementen mit dem Element **<LineBreak />** (das abschließende Slash-Zeichen ist zwingend erforderlich):

```
<Paragraph FontSize="12" FontFamily="Arial">Text mit einem Zeilenumbruch
    <LineBreak /> vor diesem Teil.</Paragraph>
```

Abbildungen mit dem Figure-Element

Mit dem **Figure**-Element können Sie Abbildungen und andere Elemente in den Fließtext integrieren. Dazu fügen Sie in das **Paragraph**-Element ein **Figure**-Element ein. Dieses bietet Eigenschaften wie die folgenden:

- **HorizontalAnchor**: Horizontaler Anker, **ColumnRight** verankert beispielsweise am rechten Rand der aktuellen Spalte
- **VerticalAnchor**: Vertikaler Anker, **ParagraphTop** beispielsweise verankert am oberen Rand des Absatzes

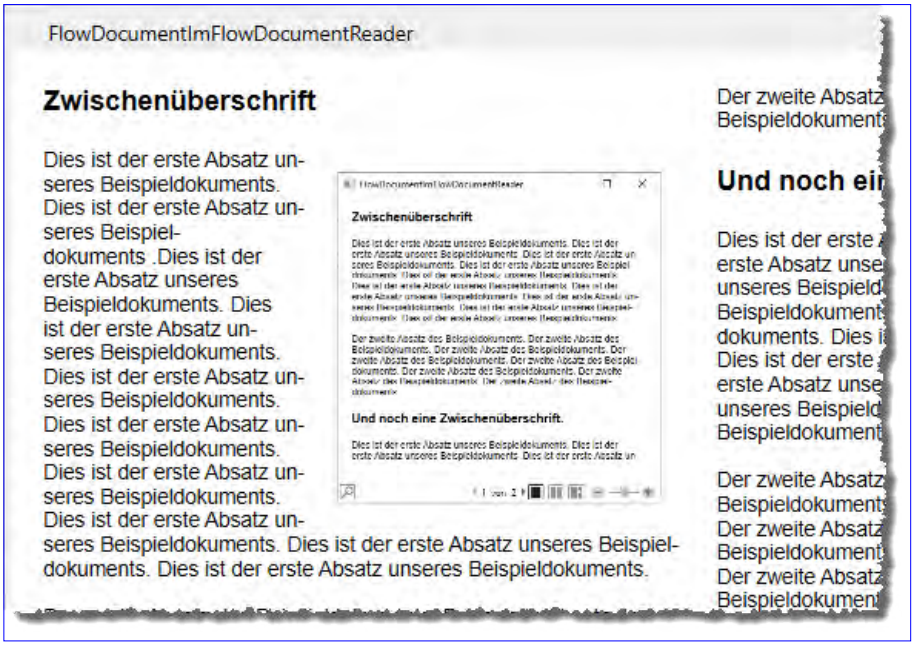


Bild 6: Einbetten einer Abbildung per **Figure**-Element

- **Width**/**Height**: Geben Breite und Höhe des Elements an.
- **VerticalOffset**: Gibt den vertikalen Versatz an.
- **HorizontalOffset**: Gibt den horizontalen Versatz an.

Mit diesen Eigenschaften lässt sich eine Abbildung wie in Bild 6 einbetten.

Dazu benötigen wir noch einige weitere Elemente, die wir in das **Figure**-Element einbauen – zunächst ein **BlockUIContainer**-Element ohne Attribute und darin den eigentlichen Inhalt, in diesem Fall ein **Image**-Element mit dem anzuzeigenden Bild:

```
<Paragraph FontSize="12" FontFamily="Arial">
    <Figure Width="200" HorizontalAnchor="ColumnRight" VerticalAnchor="ParagraphTop">
        <BlockUIContainer>
            <Image Source="images/pic005.png"/>
        </BlockUIContainer>
    </Figure>
Dies ist der erste Absatz unseres Beispieldokuments. ...</Paragraph>
```

Sie können dem Bild sogar noch eine Bildunterschrift hinzufügen. Dazu fügen Sie einfach noch ein **Paragraph**-Element mit dem gewünschten Text zwischen dem schließenden **BlockUIContainer**- und dem schließenden **Figure**-Element ein:

```
</BlockUIContainer>
    <Paragraph Padding="5">
        <Bold>Bild1: Beispielbild</Bold>
     </Paragraph>
</Figure>
```

Mit dem **Figure**-Element können Sie auch einfache Textkästen in den Fließtext integrieren – siehe Bild 7. Im folgenden Beispielcode haben wir dazu ein neues **Figure**-Element in das **Paragraph**-Element eingefügt, das den Kasten enthalten soll. Für das **Figure**-Element haben wir die Attribute so eingestellt, dass der Kasten rechts oben in der Textspalte mit dem entsprechenden Absatz verankert wird. Außerdem haben wir die Breite auf **200** festgelegt, einen

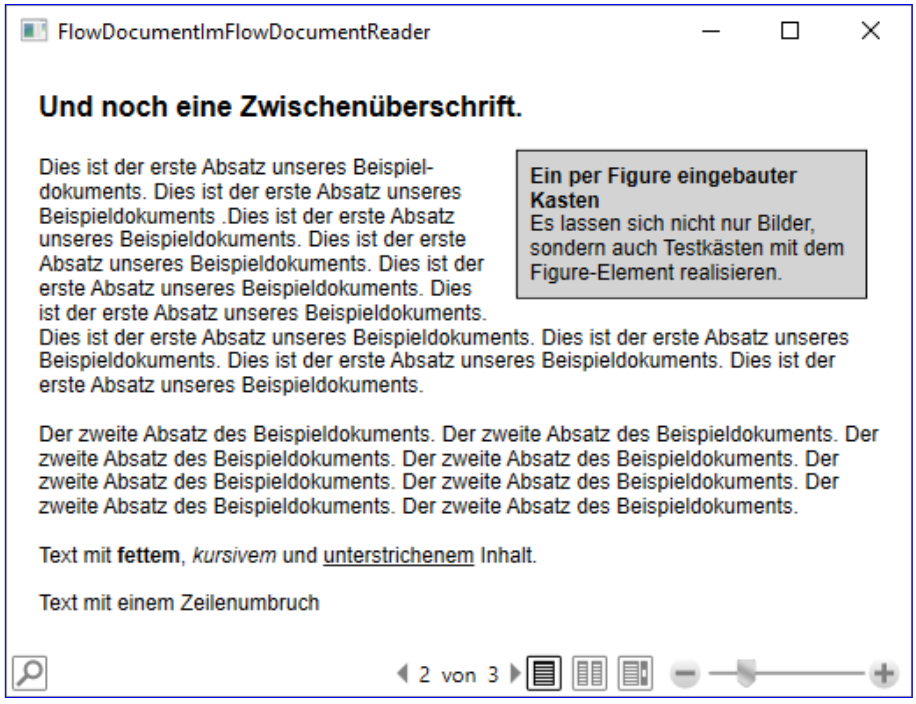


Bild 7: Einbetten eines Kastens mit Text in den Fließtext

Rahmen definiert und einen grauen Hintergrund. Im Kasten haben wir dann eine fett gesetzte Überschrift sowie einen normal gesetzten Text eingefügt:

```
<Paragraph FontSize="12" FontFamily="Arial">
    <Figure Width="200" VerticalOffset="-10" BorderBrush="Black" BorderThickness="1" Background="LightGray">
        <Paragraph FontWeight="Bold">Ein per Figure eingebauter Kasten</Paragraph>
        <Paragraph>Es lassen sich nicht nur Bilder, sondern auch Testkästen mit dem Figure-Element realisieren.</Paragraph>
    </Figure>Dies ist der erste Absatz unseres Beispieldokuments. ...
</Paragraph>
```

Weitere Formatierung mit dem Span-Element

Mit den drei Inline-Elementen **Bold**, **Italic** und **Underline** können Sie zwar bereits einige Hervorhebungen im Text unterbringen, aber manchmal bedarf es etwas mehr Anpassungen. Dazu können Sie das **Span**-Element nutzen. Dieses verwenden Sie prinzipiell wie unter HTML: Sie fassen den betroffenen Text in ein öffnendes und ein schließendes **Span**-Element ein und versehen das öffnende Element dann mit den gewünschten Attributen. Das sieht beispielsweise so aus:

```
<Paragraph FontSize="12" FontFamily="Arial">Der zweite Absatz des <Span FontWeight="Black" Foreground="Red">Beispieldokum
ents</Span>. Der zweite Absatz des Beispieldokuments. ...</Paragraph>
```

Sie können auch Text wie Hyperlinks formatieren und diese mit einer Link-Funktion versehen. Dazu nutzen Sie das **Hyperlink**-Element:

```
<Paragraph FontFamily="Arial" FontSize="12">Absatz mit URL:
    <Hyperlink>http://datenbankentwickler.net</Hyperlink></Paragraph>
```

Weitere Blöcke

Neben dem Absatz beziehungsweise dem **Paragraph**-Element als vermutlich am häufigsten verwendeten Element gibt es noch vier weitere Elemente:

- **Section**: Kann weitere Elemente wie **BlockUIContainer**, **List**, **Paragraph**, **Table** und auch weitere **Section**-Elemente aufnehmen und beispielsweise Attribute für alle untergeordneten Elemente festlegen.
- **List**: Erlaubt das Formatieren von Absätzen in Listenform.
- **Table**: Ermöglicht das Strukturieren von Texten und Elementen in Tabellen.
- **BlockUIContainer**: Mit dem **BlockUIContainer**-Element können Sie WPF-Steuerelemente in das FlowDocument einfügen.

Das RichTextBox-Steuerelement

Die beiden Steuerelemente TextBox und TextBlock reichen für die Darstellung einfacher Texte aus. Wenn die Texte jedoch formatiert oder sogar strukturiert dargestellt werden wollen, brauchen wir eine Alternative. Im Artikel »FlowDocument-Elemente mit XAML« haben Sie bereits das FlowDocument-Element kennen gelernt, das viele Möglichkeiten bietet einschließlich Absätze, Inline-Formatierungen, Listen, Tabellen und Abbildungen. Nun wollen wir uns ansehen, wie uns das RichTextBox-Steuerelement dabei unterstützen kann, diese Art von Dokumenten zu bearbeiten.

FlowDocument als Inhalt von RichTextBox-Steuerelementen

RichTextBox-Steuerelemente enthalten immer ein **FlowDocument**-Element als Inhalt. Wenn Sie also Text in ein solches Steuerlement eingeben, dann wird dieser Text im Hintergrund als Inhalt eines **FlowDocument**-Elements gespeichert. Ein **FlowDocument**, das haben wir bereits im Artikel **FlowDocument**-Elemente mit XAML beschrieben, ist im Prinzip ein XML-Dokument, das sowohl den Inhalt also auch die Formatierungen und die Strukturierung des Dokuments enthält.

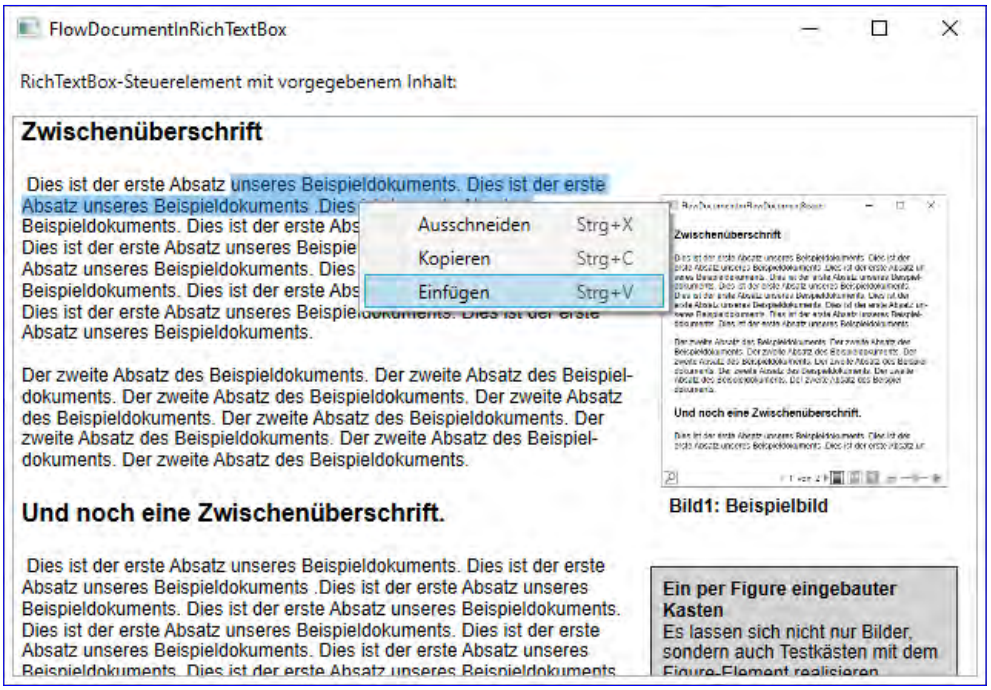


Bild1: Beispielbild

Bild 1: RichTextBox-Element mit formatiertem Text, der sich aber nicht formatieren lässt

Wir wollen ein Fenster erzeugen, dass in einem Grid ein Label und ein **RichTextBox**-Steuerelement enthält und den Inhalt eines der **FlowDocument**-Elemente, die wir im oben genannten Artikel erzeugt haben, enthält (siehe Bild 1).

Dazu nutzen wir den folgenden Code:

```
<Grid>
    <Grid.RowDefinitions>
        <RowDefinition Height="Auto"></RowDefinition>
        <RowDefinition Height="*"></RowDefinition>
```

```
    </Grid.RowDefinitions>
    <Label Grid.Row="0" Margin="5">RichTextBox-Steuerelement mit vorgegebenem Inhalt:</Label>
    <RichTextBox Grid.Row="1" Margin="5">
        <FlowDocument IsOptimalParagraphEnabled="true" IsHyphenationEnabled="True" TextAlignment="Left">
            <Paragraph FontSize="16" FontFamily="Arial" FontWeight="Bold">Zwischenüberschrift</Paragraph>
            ...
        </FlowDocument>
    </RichTextBox>
</Grid>
```

Wir haben hier direkt den Beispielcode aus dem Artikel **FlowDocument-Elemente mit XAML** (**www.datenbankentwickler.net/203**) in den Code integriert, damit wir direkt Text zum Herumprobieren haben. Im Artikel **FlowDocument-Elemente mit VB** (**www.datenbankentwickler.net/210**) zeigen wir, wie Sie **FlowDocument**-Elemente per VB mit Inhalt füllen, diese auslesen oder bearbeiten.

Leider zeigt das **RichTextBox**-Element in der aktuellen Konfiguration keine Steuerelemente zum Formatieren des Textes an – zunächst keine offensichtlichen, aber auch keine über das Kontextmenü. Außerdem sehen wir in der Abbildung auch direkt, dass sich das **RichTextBox**-Element deutlich von den Steuerelementen zur Anzeige von **FlowDocument**-Elementen unterscheidet. Der Inhalt wird immer in einer Spalte angezeigt, es gibt keine Zoom- und auch keine Suchfunktion. Wir haben es also mit einem Steuerelement rein zum Bearbeiten des Inhalts zu tun. Damit Sie den enthaltenen Text auch in gewohnter Form bearbeiten können, fügen wir im Artikel **Format-Menü für die RichTextBox** (**www.datenbankentwickler.net/205**) ein passendes Menü hinzu.

RichTextBox-Steuerelement leeren

Wenn Sie mit einem leeren **RichTextBox**-Steuerelement fortfahren wollen, können Sie das schnell erledigen, indem Sie den kompletten Inhalt markieren (mit **Strg + A**) und dann die **Entf**-Taste betätigen. Sie finden dann ein leeres **RichTextBox**-Steuerelement vor (siehe Bild 2).

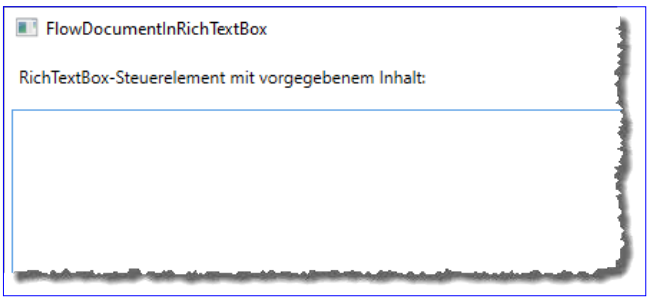


Bild 2: Geleertes **RichTextBox**-Steuerelement

Kompletten Inhalt markieren

Wir wollen zeigen, wie Sie per VB den kompletten Inhalt des **RichTextBox**-Steuerelements markieren. Dazu fügen wir eine erste Schaltfläche oberhalb des **RichTextBox**-Steuerelements hinzu, das wir wie folgt in ein **StackPanel**-Element mit horizontaler Ausrichtung zusammenfassen:

```
<StackPanel Orientation="Horizontal" Grid.Row="1">
    <Button x:Name="btnAllesMarkieren" Click="BtnAllesMarkieren_Click">Alles markieren</Button>
</StackPanel>
```

Die Ereignismethode enthält den Aufruf der **SelectAll**-Methode des **RichTextBox**-Steuerelements sowie die Anweisung zum Verschieben des Fokus auf dieses Steuerelement (siehe Bild 3).

Die **SelectAll**-Methode reicht zwar aus, um den Inhalt des **RichTextBox**-Steuerelements zu markieren, aber solange das Steuerelement nicht den Fokus hat, wird die Markierung nicht angezeigt. Daher müssen wir noch den Fokus auf das **RichTextBox**-Element verschieben.

```
Private Sub BtnAllesMarkieren_
Click(...)
    rtb.SelectAll()
    rtb.Focus()
End Sub
```

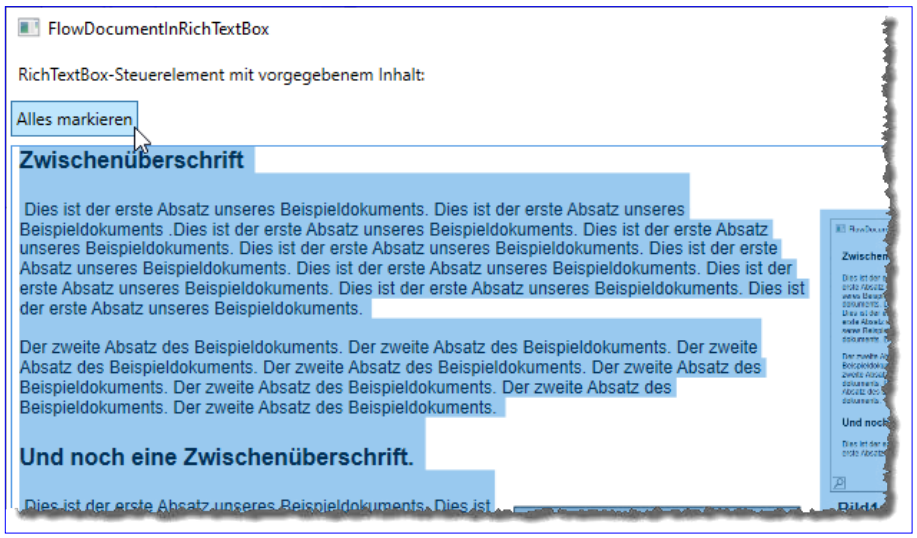


Bild 3: Markieren des vollständigen Inhalts des **RichTextBox**-Steuerelements

Text ausgeben

Um den vollständigen Text auszugeben, müssen wir einen etwas anderen Weg gehen als etwa bei einem Textfeld. Hier reicht es, über das **Text**-Attribut auf den Inhalt zuzugreifen.

Dazu deklarieren wir ein **TextRange**-Objekt namens **objTextRange** und füllen dieses mit einem neuen Objekt dieses Typs mit den Konstruktor-Parametern **rtb.Document.ContentStart** und **rtb.Document.ContentEnd**. Dann geben wir den Inhalt von **objTextRange** über die **Text**-Eigenschaft in einem Meldungsfenster aus:

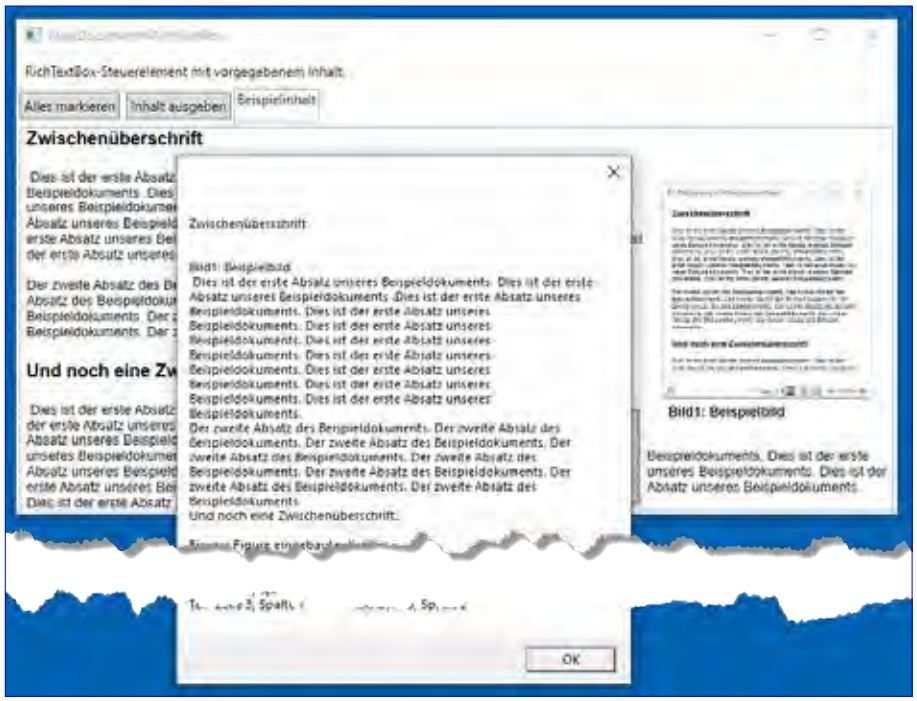


Bild 4: Ausgabe des Inhalts des **RichTextBox**-Steuerelements

```
Private Sub BtnInhaltAusgeben_Click(sender As Object, e As RoutedEventArgs)
    Dim objTextRange As TextRange
    objTextRange = New TextRange(rtb.Document.ContentStart, rtb.Document.ContentEnd)
    MessageBox.Show(objTextRange.Text)
End Sub
```

Das Ergebnis zeigt Bild 4. Hier ist zu erkennen, dass die Ausgabe ohne Formatierungen geliefert wird.

Markierten Text auslesen

Den aktuell markierten Text ermitteln wir mit der **Selection**-Eigenschaft des **RichTextBox**-Steuerelements. Dieses bietet mit der **Text**-Eigenschaft die Möglichkeit, den enthaltenen Text auszulesen. Die folgende Methode gibt den markierten Text aus:

```
Private Sub BtnMarkierterText_Click(sender As Object, e As RoutedEventArgs)
    Dim strSelection As String
    strSelection = rtb.Selection.Text
    MessageBox.Show(strSelection)
End Sub
```

Auf Änderung der Markierung reagieren

Manchmal will man auf das Setzen oder Ändern einer Markierung reagieren. Dazu gibt es ein passendes Ereignis, das wir durch Setzen des Attributs **SelectionChanged** auf den Namen der auszuführenden Ereignismethode aktivieren:

```
<RichTextBox x:Name="rtb" ... SelectionChanged="Rtb_SelectionChanged">
```

Das **Selection**-Ereignis feuert mehrmals, wenn Sie beispielsweise mit der Maus einen Text markieren und dazu an einer Stelle die Maustaste herunterdrücken und dann den Mauszeiger bis zum Ende der Markierung ziehen oder die Markierung mithilfe von Tastaturbefehlen erweitern. Daher geben wir den aktuell markierten Text im folgenden Beispiel in einem Textfeld aus:

```
Private Sub Rtb_SelectionChanged(sender As Object, e As RoutedEventArgs)
    Dim strSelection As String
    strSelection = rtb.Selection.Text
    txt.Text = strSelection
End Sub
```

Text an Markierung einfügen

Die folgende Schaltfläche soll eine Methode aufrufen, die den Text aus dem Textfeld an der Stelle der aktuellen Markierung im **RichTextBox**-Steuerelement einfügen soll:

```
<Button x:Name="btnTextEinfuegen" Click="BtnTextEinfuegen_Click">Text an Markierung einfügen:</Button>
<TextBox x:Name="txtEinfuegen" Text="Einfügen"></TextBox>
```

Die dadurch aufgerufene Ereignismethode erledigt das durch Einstellen des mit **Selection.Text** ermittelten Bereichs auf den Inhalt des Textfeldes:

```
Private Sub BtnTextEinfuegen_Click(sender As Object, e As RoutedEventArgs)
    Dim strText As String
    strText = txtEinfuegen.Text
    rtb.Selection.Text = strText
End Sub
```

Kompletten Inhalt durch neuen Text ersetzen

Wenn Sie den kompletten aktuellen Inhalt des **RichTextBox**-Steuerelements durch einen anderen Text ersetzen wollen, legen Sie den zu ersetzenden Teil mit einem **TextRange**-Element fest. Dieses füttern wir beim Initialisieren mit der Start- und der Endmarkierung des enthaltenen Dokuments und weisen dann der **Text**-Eigenschaft des **TextRange**-Objekts den einzusetzenden Text zu:

```
Private Sub BtnKomplettenTextErsetzen_Click(sender As Object, e As RoutedEventArgs)
    Dim objTextRange As TextRange
    Dim strText As String
    strText = txtEinfuegen.Text
    objTextRange = New TextRange(rtb.Document.ContentStart, rtb.Document.ContentEnd)
    objTextRange.Text = strText
End Sub
```

Texte speichern und laden

Wenn Sie eine kleine Textverarbeitung bauen wollen, möchten Sie die eingegebenen Texte auch speichern und wieder in das **RichTextBox**-Steuerelement laden können. Als Erstes wollen wir den aktuell im **RichTextBox**-Steuerelement enthaltenen Text in einer Datei speichern. Dazu benötigen wir einen Dialog zum Auswählen des Pfades der Datei, den wir mit dieser Funktion aufrufen:

```
Private Function GetSaveFilename() As String
    Dim objSaveFileDialog As SaveFileDialog
    Dim strFilename As String = ""
    objSaveFileDialog = New SaveFileDialog
    If (objSaveFileDialog.ShowDialog = True) Then
        strFilename = objSaveFileDialog.FileName
    End If
    Return strFilename
End Function
```

Diese Funktion nutzen wir in der Ereignismethode, die durch die Schaltfläche **btnSpeichern** ausgelöst wird. Hier ermitteln wir zunächst den Dateinamen mit der Funktion **SaveFilename**:

```
Private Sub BtnSpeichern_Click(sender As Object, e As RoutedEventArgs)
    Dim objTextRange As TextRange
    Dim objFileStream As FileStream
    Dim strDateiname As String
    strDateiname = GetSaveFilename()
```

Dann prüfen wir, ob ein Dateiname zum Speichern eingegeben wurde. Falls ja, erstellen wir ein neues **TextRange**-Objekt, das mit dem wir den kompletten Inhalt des **RichTextBox**-Steuerelements füllen. Dann erstellen wir ein **FileStream**-Objekt, das ein

neue Datei auf Basis des ermittelten Pfades und dem Modus **Create** anlegt. Mit der **Save**-Methode des **TextRange**-Objekts speichern wir den Inhalt im angegebenen **FileStream**-Objekt mit dem Datenformat **XamlPackage**:

```
    If Not strDateiname.Length = 0 Then
        objTextRange = New TextRange(rtb.Document.ContentEnd, rtb.Document.ContentStart)
        objFileStream = New FileStream(strDateiname, FileMode.Create)
        objTextRange.Save(objFileStream, DataFormats.XamlPackage)
        objFileStream.Close()
    End If
End Sub
```

Schauen wir uns nun das Ergebnis im Texteditor an, erhalten wir das Bild aus Bild 5. Das Speicherformat **DataFormats.XamlPacke** ist also definitiv kein Format, das für das menschliche Auge bestimmt ist. Gleich wollen wir noch ein alternatives Format wählen, um zu sehen, ob wir den Inhalt auch im XAML-Format so speichern können, wie wir das Dokument auch im XAML-Editor programmiert haben.

Erst einmal wollen wir das Dokument jedoch wieder in das **RichTextBox**-Steuerelement einlesen. Dazu definieren wir zunächst eine Funktion zum Auswählen der zu öffnenden Datei:

```
Private Function GetOpenFilename() As String
    Dim objOpenFileDialog As OpenFileDialog
    Dim strFilename As String = ""
    objOpenFileDialog = New OpenFileDialog
    If (objOpenFileDialog.ShowDialog() = True) Then
        strFilename = objOpenFileDialog.FileName
    End If
    Return strFilename
End Function
```

Die **Click**-Ereignismethode für die Schaltfläche **BtnOeffnen** ermittelt über die Funktion **GetOpenFileName** den Namen der einzulesenden Datei:

```
Private Sub BtnOeffnen_Click(sender _
        As Object, e As RoutedEventArgs)
    Dim objTextrange As TextRange
    Dim strDateiname As String = ""
    Dim objFileStream As FileStream
    strDateiname = GetOpenFilename()
```

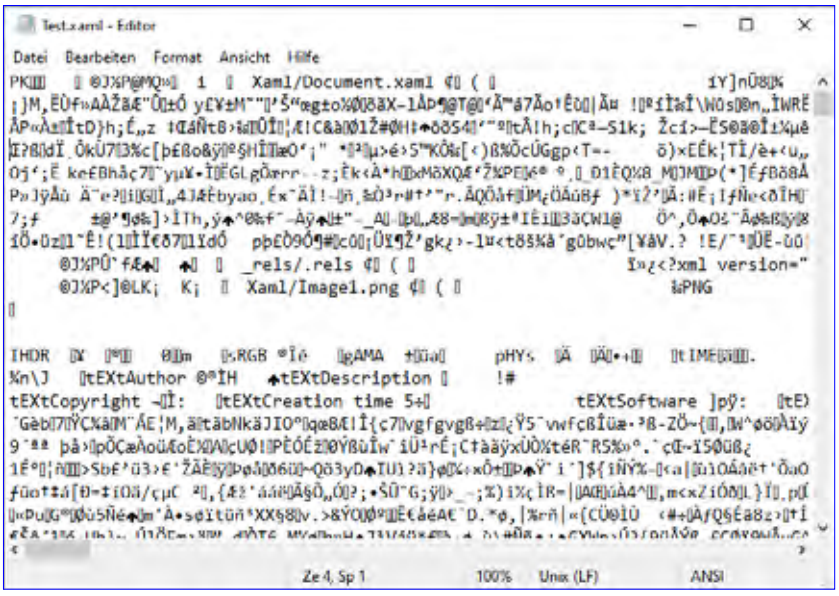


Bild 5: Gespeichertes **FlowDocument**

Format-Menü für die RichTextBox

Das RichTextBox-Steuerelement unter WPF offeriert eine Reihe von Formatierungsmöglichkeiten. Damit Sie diese Formatierungen komfortabel anwenden können, bauen wir in diesem Artikel ein Menü mit den gängigsten Befehlen zur Formatierung von Texten im RichTextBox-Steuerelement. Dabei können wir gleich einige der Grundlagen, die wir im Artikel Das RichTextBox-Steuerelement kennengelernt haben, praktisch umsetzen.

RichTextBox-Inhalte formatieren

Da das RichTextBox-Steuerelement keine offensichtlichen Steuerelemente zum Formatieren bietet, legen wir diese einfach selbst an. Das Ergebnis soll anschließend wie in Bild 1 aussehen. Wir wollen also Schaltflächen zum Ausrichten des Absatzes, zum Einstellen fetter, kursiver und unterstrichener Schrift, zum Umwandeln von Absätzen in Auflistungen oder nummerierte Listen sowie zum Einstellen der Schriftart und -größe vorsehen.

Die Steuerelemente platzieren wir in einer zusätzlichen Zeile des **Grid**-Elements. Damit wir nicht alle Befehle mit Texten ausstatten müssen und da es überdies für die meisten gängigen Formatierungsbefehle gängige Icons gibt, statten wir auch unsere Schaltflächen mit passenden Icons aus. Diese fügen wir dem Unterverzeichnis **images** unseres Projekts hinzu (siehe Bild 2).

Alternative Icons direkt von Microsoft finden Sie etwa unter folgendem Link:

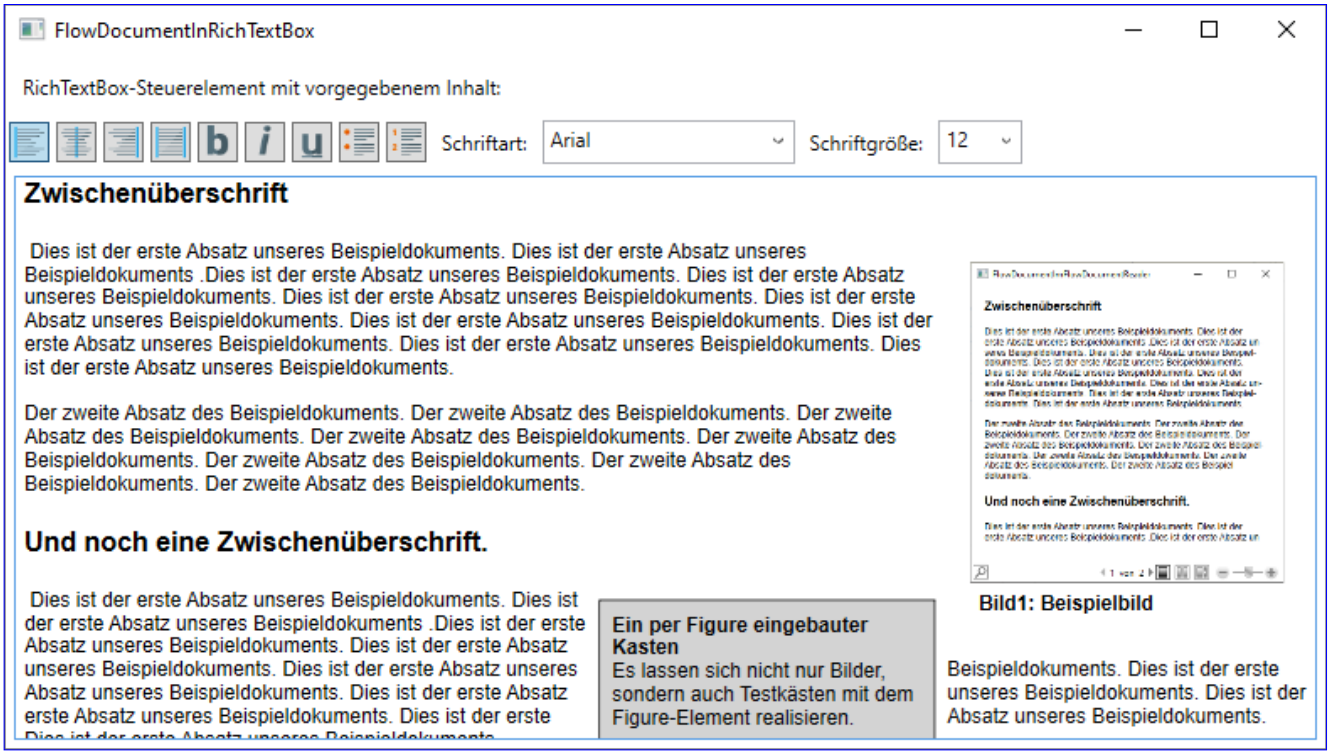


Bild 1: RichTextBox-Steuerelement mit Formatierungs-Menü

https://www.microsoft.com/en-us/download/details.aspx?id=35825

Wir verwenden jedoch die Icons von der Seite **www.iconexperience.com**.

Der neuen Zeile im **Grid**-Element fügen wie ein **StackPanel**-Element mit horizontaler Ausrichtung hinzu, das vier **Button**-Elemente aufnimmt. Diese statten wir jeweils mit einer Bezeichnung wie **btnTextAlignLeft**, **btnTextAlignCenter**, **btnTextAlignRight** und **btnTextAlignJustified** aus und legen für das **Click**-Ereignis einer jeden Schaltfläche eine passende Ereignismethode an:

```
<StackPanel Orientation="Horizontal" Grid.Row="1">
    <Button x:Name="btnTextAlignLeft" Margin="2" Width="24" Height="24" HorizontalAlignment="Left"
            Click="BtnTextAlignLeft_Click">
        <Image Source="images/text_align_left.ico"></Image>
    </Button>
    <Button x:Name="btnTextAlignCenter" Margin="2" Width="24" Height="24" HorizontalAlignment="Left"
            Click="BtnTextAlignCenter_Click">
        <Image Source="images/text_align_center.ico"></Image>
    </Button>
    <Button x:Name="btnTextAlignRight" Margin="2" Width="24" Height="24" HorizontalAlignment="Left"
            Click="BtnTextAlignRight_Click">
        <Image Source="images/text_align_right.ico"></Image>
    </Button>
    <Button x:Name="btnTextAlignJustified" Margin="2" Width="24" Height="24"
            HorizontalAlignment="Left" Click="BtnTextAlignJustified_Click">
        <Image Source="images/text_align_justified.ico"></Image>
    </Button>
</StackPanel>
```

Die dadurch ausgelösten Methoden implementieren wir im Code behind-Modul des XAML-Fensters:

```
Public Class FlowDocumentInRichTextBox
    Private Sub BtnTextAlignCenter_Click(sender As Object, _
            e As RoutedEventArgs)
        EditingCommands.AlignCenter.Execute(vbNull, Me.rtb)
    End Sub

    Private Sub BtnTextAlignJustified_Click(sender As Object, _
            e As RoutedEventArgs)
        EditingCommands.AlignJustify.Execute(vbNull, Me.rtb)
    End Sub
```

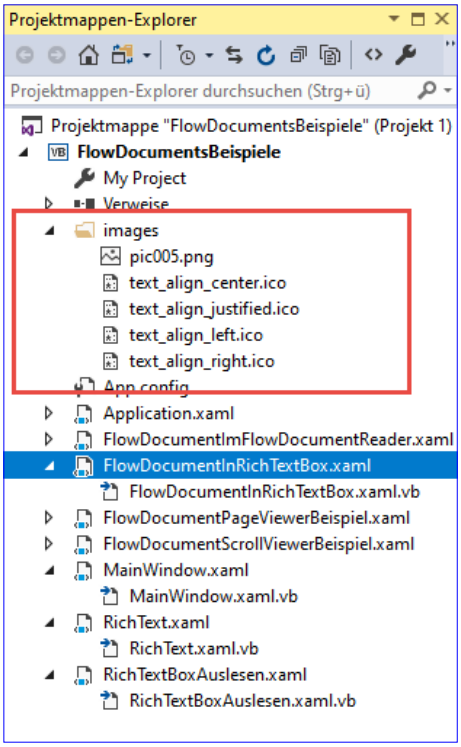


Bild 2: Icons im Projektmappen-Explorer

```
    Private Sub BtnTextAlignLeft_Click(sender As Object, e As RoutedEventArgs)
        EditingCommands.AlignLeft.Execute(vbNull, Me.rtb)
    End Sub

    Private Sub BtnTextAlignRight_Click(sender As Object, e As RoutedEventArgs)
        EditingCommands.AlignRight.Execute(vbNull, Me.rtb)
    End Sub
End Class
```

Die Methoden rufen jeweils eine Methode der Klasse **EditingCommands** auf, in diesem Fall **AlignCenter**, **AlignJustify**, **AlignLeft** und **AlignRight**. Diese wird mit der **Execute**-Methode ausgeführt, wobei wir mit dem zweiten Parameter dieser Methode das Ziel der Methode angeben – in diesem Fall das **RichTextBox**-Element.

Das gleiche Ergebnis erhalten wir übrigens, wenn wir die entsprechenden **Command**-Elemente dem Attribut **Command** der Schaltflächen zuweisen. Dazu fügen wir einen neuen Satz mit vier weiteren Schaltflächen zum **.xaml**-Code hinzu. Jede Schaltfläche erhält das Attribut **Command** mit dem Befehl, den wir zuvor in der Ereignisprozedur aufgerufen haben – zum Beispiel **EditingCommands.AlignLeft**.

Das allein bewirkt allerdings noch nichts, wenn wir nicht angeben, auf welches Steuerelement sich die Aktion auswirken soll. Dazu stellen wir das Attribut **CommandTarget** jeweils auf den Wert **{Binding ElementName=rtb}** ein, wobei **rtb** der Name des **RichTextBox**-Steuerelements ist:

```
<Button x:Name="btnTextAlignLeftCommand" Margin="2" Width="24" Height="24" HorizontalAlignment="Left"
Command="EditingCommands.AlignLeft" CommandTarget="{Binding ElementName=rtb}">
    <Image Source="images/text_align_left.ico"></Image>
</Button>
<Button x:Name="btnTextAlignCenterCommand" Margin="2" Width="24" Height="24" HorizontalAlignment="Left"
Command="EditingCommands.AlignCenter" CommandTarget="{Binding ElementName=rtb}">
    <Image Source="images/text_align_center.ico"></Image>
</Button>
<Button x:Name="btnTextAlignRightCommand" Margin="2" Width="24" Height="24" HorizontalAlignment="Left"
Command="EditingCommands.AlignRight" CommandTarget="{Binding ElementName=rtb}">
    <Image Source="images/text_align_right.ico"></Image>
</Button>
<Button x:Name="btnTextAlignJustifiedCommand" Margin="2" Width="24" Height="24" HorizontalAlignment="Left"
Command="EditingCommands.AlignJustify" CommandTarget="{Binding ElementName=rtb}">
    <Image Source="images/text_align_justified.ico"></Image>
</Button>
```

Das Ergebnis sieht wie in Bild 3 aus (hier zunächst nur mit dem ersten Satz der vier Schaltflächen – die übrigen ergänzen wir auf den folgenden Seiten).

Damit Sie schnell im **RichTextBox**-Element arbeiten können, können Sie die meisten Befehle auch per eingebauter Tastenkombination aufrufen – für die wichtigsten drei der vier oben vorgestellten Befehle beispielsweise mit den folgenden Tastenkombinationen:

- **AlignCenter** (**Strg + E**): Zentrierte horizontale Ausrichtung
- **AlignLeft** (**Strg + L**): Linkszentrierte horizontale Ausrichtung
- **AlignRight** (**Strg + R**): Rechtszentrierte horizontale Ausrichtung

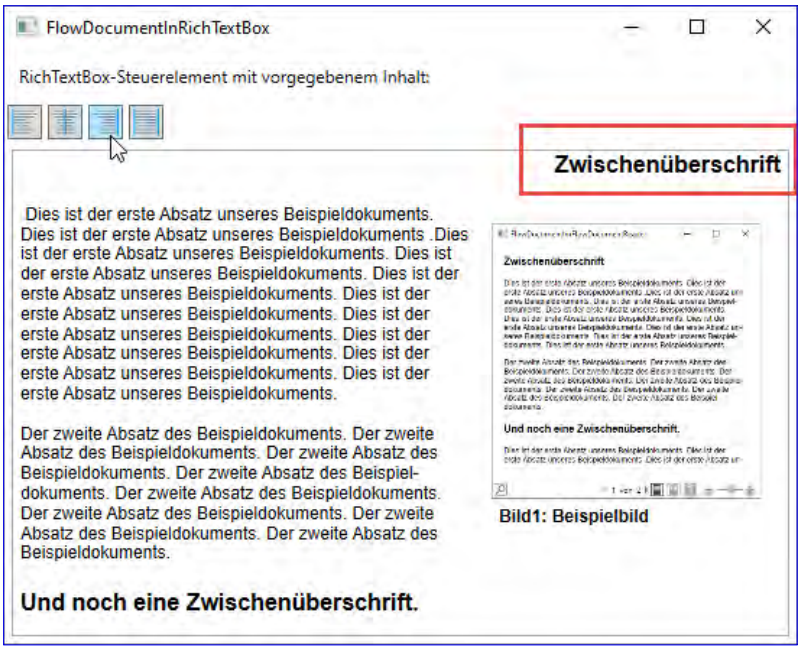


Bild 3: Ausrichten von Text mit selbstgebauten Schaltflächen

Die übrigen Befehle und Tastenkombinationen können Sie beispielsweise dieser Seite entnehmen:

```
https://docs.microsoft.com/en-us/dotnet/api/system.windows.documents.editingcommands?view=netcore-3.1
```

Neben den vier Schaltflächen zum Einstellen der Ausrichtung des aktuellen Absatzes benötigen wir noch weitere Steuerelemente. Zunächst wollen wir uns um die Steuerelemente zum Formatieren von Text kümmern (siehe Bild 4).

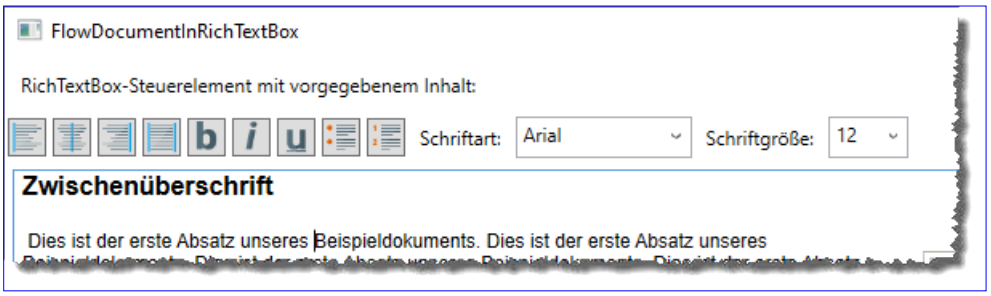


Bild 4: Weitere Steuerelemente zum Formatieren von Text

Diese fügen wir wie folgt hinzu und statten diese mit den entsprechenden **Command**-Befehlen aus. Wir starten mit den drei Schaltflächen für die fette, kursive und unterstrichene Zeichenformatierung, die wir mit folgendem XAML-Code definieren:

```
<StackPanel Orientation="Horizontal" Grid.Row="1">
    ...
    <Button x:Name="btnTextBold" Margin="2" Width="24" Height="24" HorizontalAlignment="Left"
            Command="EditingCommands.ToggleBold" CommandTarget="{Binding ElementName=rtb}">
        <Image Source="images/font_style_bold.ico"></Image>
    </Button>
```

```
<Button x:Name="btnTextItalic" Margin="2" Width="24" Height="24" HorizontalAlignment="Left"
        Command="EditingCommands.ToggleItalic" CommandTarget="{Binding ElementName=rtb}">
    <Image Source="images/font_style_italics.ico"></Image>
</Button>
<Button x:Name="btnTextUnderline" Margin="2" Width="24" Height="24" HorizontalAlignment="Left"
        Command="EditingCommands.ToggleUnderline" CommandTarget="{Binding ElementName=rtb}">
    <Image Source="images/font_style_underline.ico"></Image>
</Button>
```

Hier sind keine umfangreichen Erläuterungen nötig – die Elemente arbeiten wie die zuvor für die Ausrichtung der Absätze angelegten Elemente. Das gleiche gilt für die folgenden beiden Schaltflächen, welche die Auflistungen und die Nummerungen ermöglichen:

```
<Button x:Name="btnBullets" Margin="2" Width="24" Height="24" HorizontalAlignment="Left"
        Command="EditingCommands.ToggleBullets" CommandTarget="{Binding ElementName=rtb}">
    <Image Source="images/list_style_bullets.ico"></Image>
</Button>
<Button x:Name="btnNumbering" Margin="2" Width="24" Height="24" HorizontalAlignment="Left"
        Command="EditingCommands.ToggleNumbering" CommandTarget="{Binding ElementName=rtb}">
    <Image Source="images/list_style_numbered.ico"></Image>
</Button>
```

Spannender wird es bei den folgenden beiden Elementen, den Kombinationsfelder zur Auswahl der Schriftart und der Schriftgröße. Zunächst das **ComboBox**-Element **cboSchriftarten**, das wir mit den Schriften des aktuellen Rechners füllen wollen. Es soll nach dem Wechseln eines Eintrags das Ereignis **SelectionChanged** auslösen, für das wir eine Ereignismethode namens **CboSchriftarten_SelectionChanged** hinterlegt haben – diese schauen wir uns gleich im Anschluss an:

```
<Label Margin="2">Schriftart:</Label>
<ComboBox x:Name="cboSchriftarten" Margin="2" Width="150" IsEditable="True"
    SelectionChanged="CboSchriftarten_SelectionChanged"></ComboBox>
```

Das **ComboBox**-Element für die Auswahl der Schriftgröße sieht ähnlich aus. Auch hier hinterlegen wir eine Ereignismethode für das Ereignis **SelectionChanged**:

```
<Label Margin="2">Schriftgröße:</Label>
<ComboBox x:Name="cboSchriftgroessen" Margin="2" Width="50" IsEditable="True"
    SelectionChanged="CboSchriftgroessen_SelectionChanged"></ComboBox>
</StackPanel>
```

Zunächst aber wollen wir die beiden Kombinationsfelder einmal füllen. In der Code behind-Klasse der **.xaml**-Datei legen wir eine Konstruktor-Methode namens **New** an, in der wir zunächst die Komponente initialisieren. Dann füllen wir die beiden

Kombinationsfelder über die **ItemsSource**-Eigenschaft. Die des **comboBox**-Elements **cboSchriftarten** füllen wir mit der Systemauflistung **System.Windows.Media.Fonts.SystemFontFamilies**, welche alle Schriften des aktuellen Systems liefert (siehe Bild 5).

Im Code sieht das wie folgt aus:

```
Public Class FlowDocumentInRichTextBox
    Public Sub New()
        InitializeComponent()
        cboSchriftarten.ItemsSource = _
System.Windows.Media.Fonts.SystemFontFamilies
```

Danach füllen wir die Eigenschaft **ItemsSource** des Kombinationsfeldes **cboSchriftgroessen** mit dem Ergebnis der Property **Schriftgroessen**:

```
        cboSchriftgroessen.ItemsSource = _
            Schriftgroessen
    End Sub
```

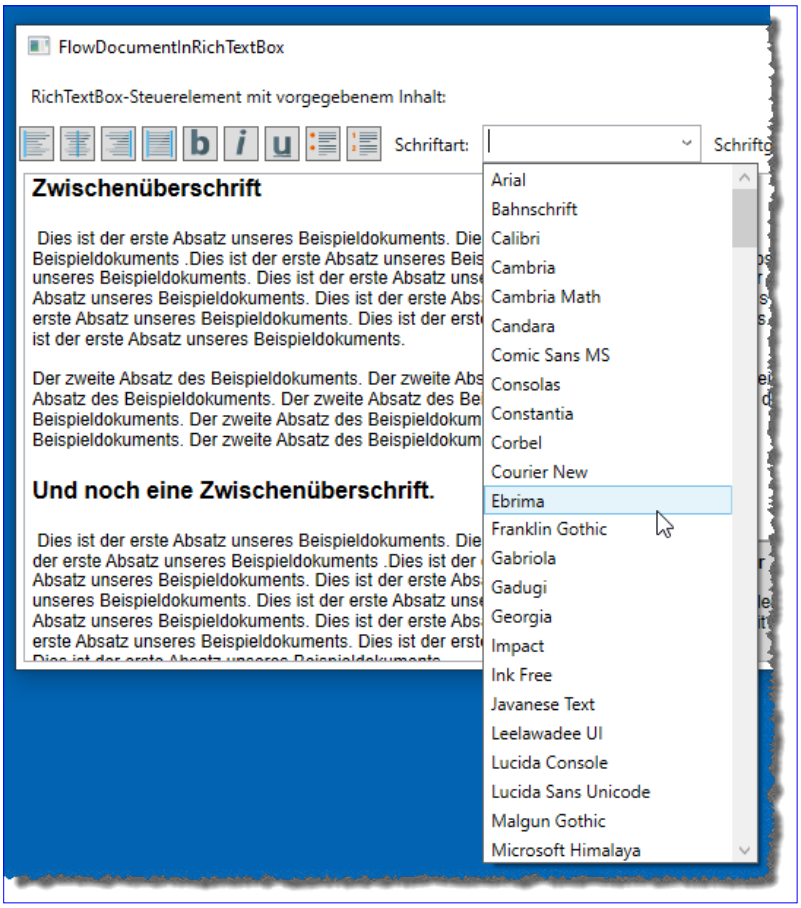


Bild 5: Einstellen der Schriftart

Diese Property deklarieren wir wie folgt. Sie liefert die gängigen Schriftgrößen:

```
    Private _schriftgroessen() As Double

    Public ReadOnly Property Schriftgroessen As Double()
        Get
            Return New Double() {6, 7, 8, 9, 10, 11, 12, 14, 16, 18, 20, 24, 28, 32, 36, 42, 48, 56, 64, 128}
        End Get
    End Property
```

Nun benötigen wir noch die Ereignismethoden, die durch das Auswählen neuer Werte in einem der beiden Kombinationsfelder **cboSchriftarten** und **cboSchriftgroessen** ausgelöst werden. Die für die Schriftart sieht wie folgt aus:

```
    Private Sub CboSchriftarten_SelectionChanged(sender As Object, e As SelectionChangedEventArgs)
        Dim objFontFamily As FontFamily = CType(e.AddedItems(0), FontFamily)
        TextformatierungEinstellen(TextElement.FontFamilyProperty, objFontFamily)
    End Sub
```

Codeschnipsel in Visual Studio

Visual Studio kommt seit einigen Versionen mit einer sehr coolen Funktion, nämlich den Code-Snippets. Diese können Sie auf verschiedene Arten in den Code einfügen. Zum Beispiel, indem Sie einen oder mehrere Buchstaben eingeben und dann die Tabulator-Taste betätigen, um den durch diese Buchstaben gekennzeichneten Codeschnipsel im Code einzufügen. In diesem Artikel schauen wir uns an, welche Codeschnipsel es schon gibt und vor allem, wie Sie selbst IntelliSense-Code-Snippets definieren können.

IntelliSense-Code-Snippets fallen Ihnen vermutlich nie auf, wenn Sie nicht über einen Tipp im Internet darüber stolpern. Deshalb wollen wir dieses spannende Thema hier aufgreifen!

Wie funktionieren diese IntelliSense-Code-Snippets also nun? Das können Sie einfach ausprobieren, indem Sie etwa in einem VB-Modul einmal die Zeichenfolge **Property** eingeben (siehe Bild 1).

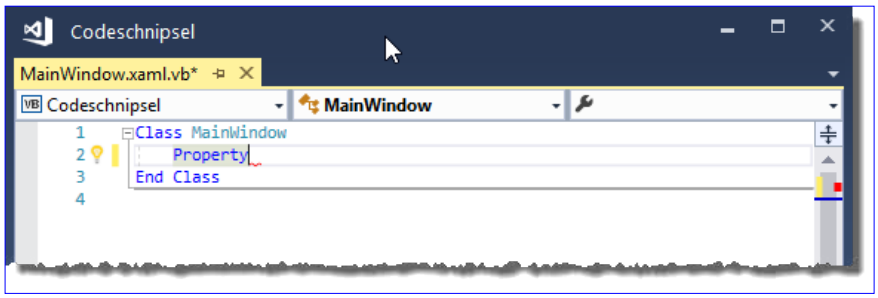


Bild 1: Vorbereiten des Code-Snippets ...

Wenn Sie dann direkt die Tabulator-Taste betätigen, wird das Schlüsselwort durch den für das Code-Snippet angegebenen Code ersetzt – siehe Bild 2.

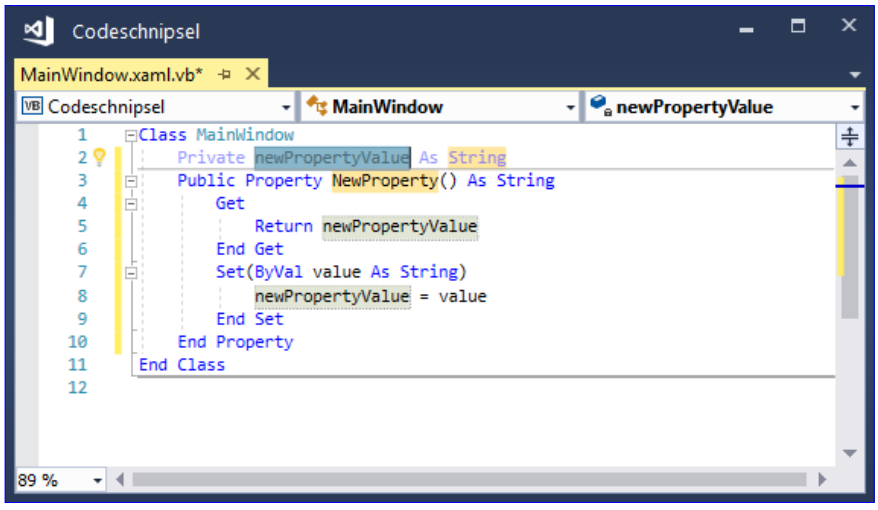


Bild 2: ... und Hinzufügen der gewünschte Code-Elemente

Weitere Einsatzmöglichkeiten von Code-Snippets

Wie und wo können wir Code-Snippets einsetzen? Es gibt zum Beispiel die folgenden Möglichkeiten:

- Einfügen von Code per Kürzel und Betätigen der Tabulator-Taste
- Betätigen der rechten Maustaste und Auswählen des zu verwendenden Codesnippets über den Kontextmenü-Eintrag **Ausschnitt\|Ausschnitt einfügen**.
- Umschließen des markierten Codes mit dem Kontextmenü-Befehl **Umschließen** (nur XAML und C#) und anschließender Eingabe des Kürzels

Umschließen des markierten Codes

Im Code möchte man gelegentlich unter VB eine oder mehrere Anweisungen in Code-Strukturen wie eine Schleife oder eine Bedingung einfassen oder unter XAML ein paar Elemente in ein übergeordnetes Element – beispielsweise, um diese in einem **StackPanel** anzuordnen. Dazu bietet Visual Studio die Möglichkeit an, den markierten Code von den gewünschten Elementen einfassen zu lassen. Unter Visual Basic funktioniert das leider nicht, sondern nur unter XAML (und unter alternativen Programmiersprachen zu Visual Basic, beispielsweise C#).

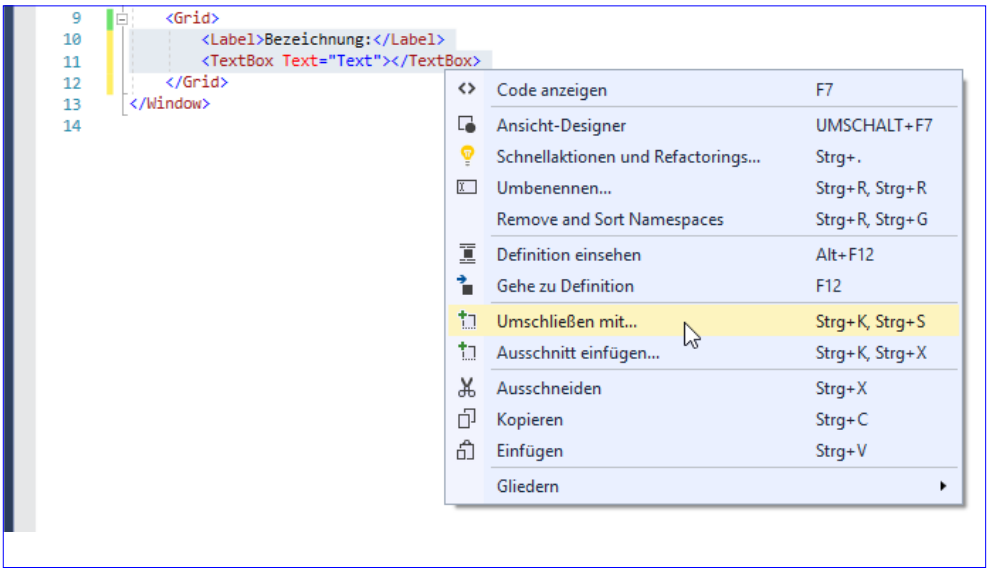


Bild 3: Umschließen von XAML-Elementen

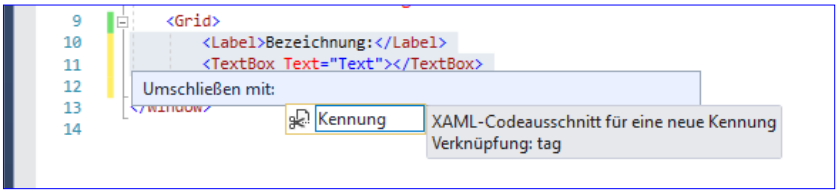


Bild 4: Auswählen des gewünschten Elements

Wenn Sie aber in XAML ein Element einfassen wollen, wie oben beschrieben durch Start- und Endtag eines **StackPanel**-Objekts, dann gehen Sie wie folgt vor:

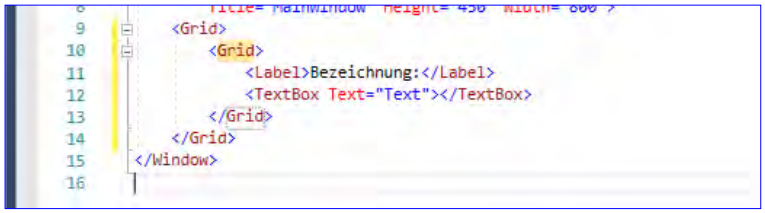


Bild 5: Der gelb markierte Text kann nun durch den Namen des gewünschten Elements ersetzt werden.

- Markieren Sie den einzufassenden Code.
- Öffnen Sie das Kontextmenü für den markierten Bereich (siehe Bild 3).
- Wählen Sie den Eintrag **Umschließen mit...** aus. Alternativ zu diesem und dem vorherigen Schritt betätigen Sie die Tastenkombination **Strg + K**, **Strg + S** (**Strg** gedrückt halten und **K** und **S** nacheinander eingeben).

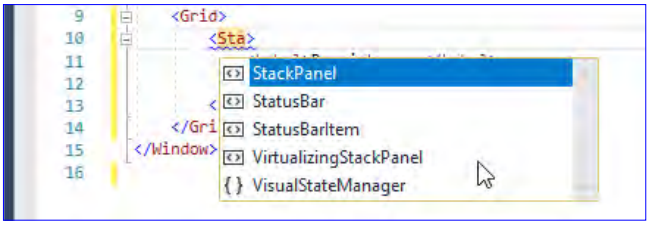


Bild 6: Beim Ersetzen unterstützt dann IntelliSense wieder.

- Es erscheint dann das Eingabefeld **Kennung**. Hier geben Sie die Kennung **tag** ein, denn es gibt nur dieses eine Element (siehe Bild 4).
- Danach wird der Name des öffnenden Elements allerdings mit gelbem Hintergrund angezeigt. Sie können dann den tatsächlich gewünschten Namen dort eingeben und die Bezeichnung **Grid** dadurch ersetzen (siehe Bild 5).

- Beim Ersetzen der markierten Elementbezeichnung hilft dann wieder IntelliSense (siehe Bild 6).

Code-Snippets verwalten

Die Code-Snippets können Sie in einem Dialog namens **Codeausschnitt-Manager** verwalten, den Sie über den Menüeintrag **Extras|Codeausschnitt**-Manager verwalten oder mit der Tastenkombination **Strg + K**, **Strg + B** öffnen.

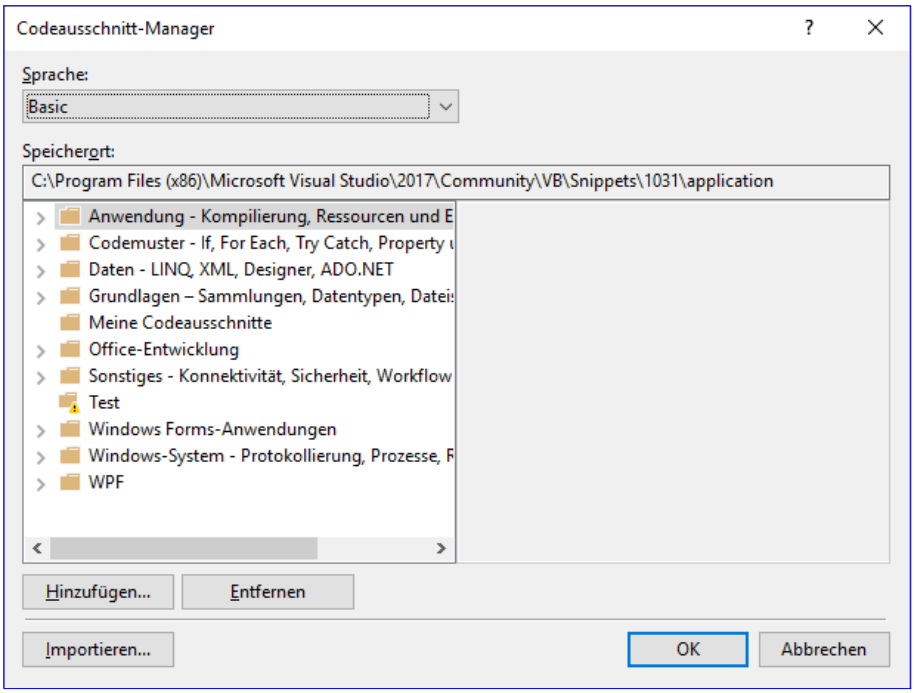


Bild 7: Der Codeausschnitt-Manager direkt nach dem Öffnen

Der Codeausschnitt-Manager sieht direkt nach dem Öffnen wie in Bild 7 aus. Sie finden hier zunächst ein Auswahlfeld für die Auswahl der Sprache, für welche die Code-Snippets angezeigt werden sollen. Der Codeausschnitt-Manager stellt direkt die aktuell verwendete Sprache ein, in diesem Fall **Basic** (für Visual Basic).

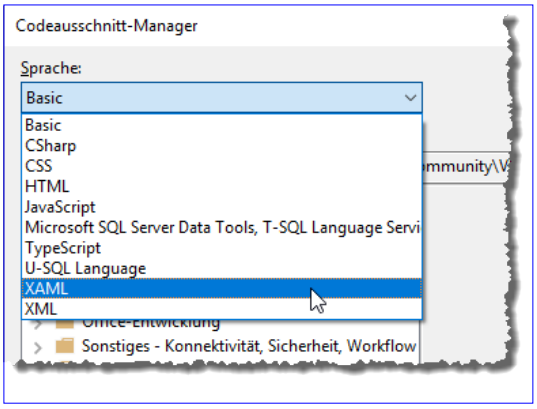


Bild 8: Auswahl der Sprache

Die Auswahl erstreckt sich von C# über Visual Basic bis hin zu **XAML** und **XML** (siehe Bild 8). Je nachdem, welche Sprache Sie auswählen, werden mehr oder weniger Ordner in der darunter befindlichen Liste angezeigt. Für XAML gibt es beispielsweise in Visual Studio 2017 erst ein Snippet namens **Kennung**.

Wenn Sie dieses Snippet auswählen, zeigt das Fenster im rechten Bereich einige Eigenschaften des Snippets an (siehe Bild 9).

Dazu gehören die folgenden Elemente:

- **Beschreibung**: Beschreibungstext zu diesem Code-Snippet
- **Verknüpfung**: Text, den Sie im Editor eingeben, um das Snippet einzufügen.

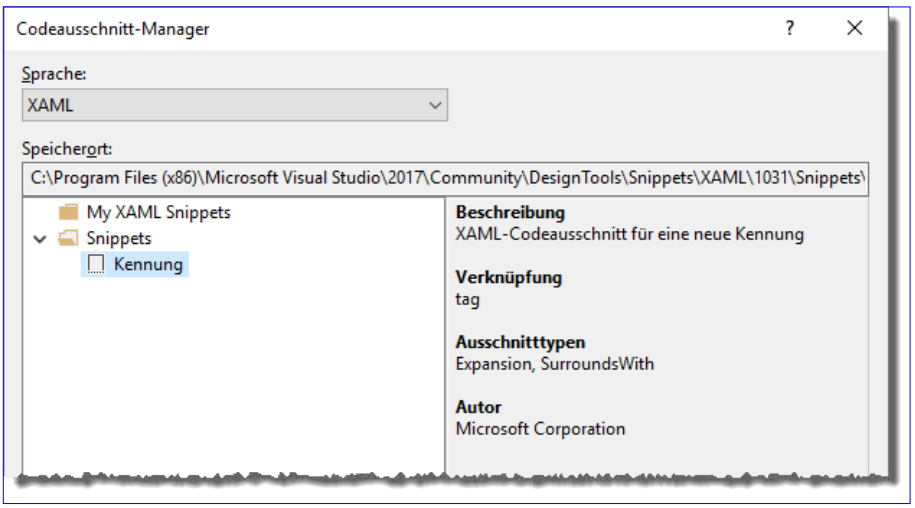


Bild 9: Informationen zu einem Snippet

- **Ausschnittstypen**: Art, wie das Code-Snippet eingefügt wird.
- **Autor**: Autor des Code-Snippets

Im unteren Bereich finden Sie noch einige Schaltflächen:

- **Hinzufügen...**: Öffnet einen Verzeichnisauswahl-Dialog. Damit können Sie alle Codesnippets dieses Verzeichnisses auf einen Rutsch importieren.
- **Entfernen**: Entfernt das aktuell angezeigten Code-Snippet aus der Sammlung der Code-Snippets.
- **Importieren...**: Öffnet einen Dateiauswahl-Dialog zur Auswahl von Dateien mit der Endung **.snippet**, die zum Codeausschnitt-Manager hinzugefügt werden sollen.

Anlegen eines eigenen Code-Snippets

Die Basis für ein neues Code-Snippet ist eine XML-Datei. Diese sieht wie folgt aus:

```
<?xml version="1.0" encoding="utf-8"?>
<CodeSnippets xmlns="http://schemas.microsoft.com/VisualStudio/2005/CodeSnippet">
    <CodeSnippet Format="1.0.0">
        <Header>
            <Title></Title>
        </Header>
        <Snippet>
            <Code Language="">
                <![CDATA[]]>
            </Code>
        </Snippet>
    </CodeSnippet>
</CodeSnippets>
```

Wir werfen einen kurzen Blick auf diese Elemente und wo Sie Informationen einfügen müssen. Die erste Lücke finden wir im Element **Title**. Hier fügen wir testweise einfach den Wert **Beispielschnipsel** ein. Als nächstes sind die Anführungszeichen des Attributes **Language** des Code-Elements leer. Hier tragen Sie die Bezeichnung der Programmiersprache ein, also etwas **CSharp** für C#, **VB** für Visual Basic oder **XAML**. Schließlich fehlt noch der einzufügende Code. Diesen tragen Sie zwischen die inneren eckigen Klammern des **CDATA**-Elements ein.

Im **Header**-Bereich können Sie noch die beiden folgenden Elemente ergänzen:

- **Author**: Gibt den Namen des Autors an.

- **Description**: Erwartet einen Beschreibungstext.

- **Shortcut**: Gibt die Abkürzung an, mit der Sie den Codeschnipsel hinzufügen können – also beispielsweise, indem Sie den Shortcut eingeben und die Tabulator-Taste betätigen.

Diese werden dann im Codeausschnitt-Manager für das entsprechende Element angezeigt.

Wir wollen ein Codesnippet für Visual Basic erstellen, das eine leere **MessageBox.Show("")**-Methode hinzufügt. Also gestalten wir das XML-Dokument wie folgt:

```
<?xml version="1.0" encoding="utf-8"?>
<CodeSnippets xmlns="http://schemas.microsoft.com/VisualStudio/2005/CodeSnippet">
    <CodeSnippet Format="1.0.0">
        <Header>
            <Title>Messagebox</Title>
            <Author>Andre Minhorst</Author>
            <Description>Fuegt eine MessageBox-Anweisung hinzu.</Description>
            <Shortcut>msg</Shortcut>
        </Header>
        <Snippet>
            <Code Language="VB">
                <![CDATA[MessageBox.Show("")]]>
            </Code>
        </Snippet>
    </CodeSnippet>
</CodeSnippets>
```

An dieser Stelle durften wir lernen, dass Umlaute oder Sonderzeichen nicht funktionieren – das Codesnippet wird dann nicht akzeptiert.

Nachdem Sie das Dokument mit der Dateiendung **.snippet** an der gewünschten Stelle gespeichert haben, können Sie es mithilfe des Codeausschnitt-Managers importieren. Dazu betätigen Sie die **Importieren**-Schaltfläche und wählen den Pfad zur **.snippet**-Datei aus. Sofern die Syntax korrekt ist, erscheint der

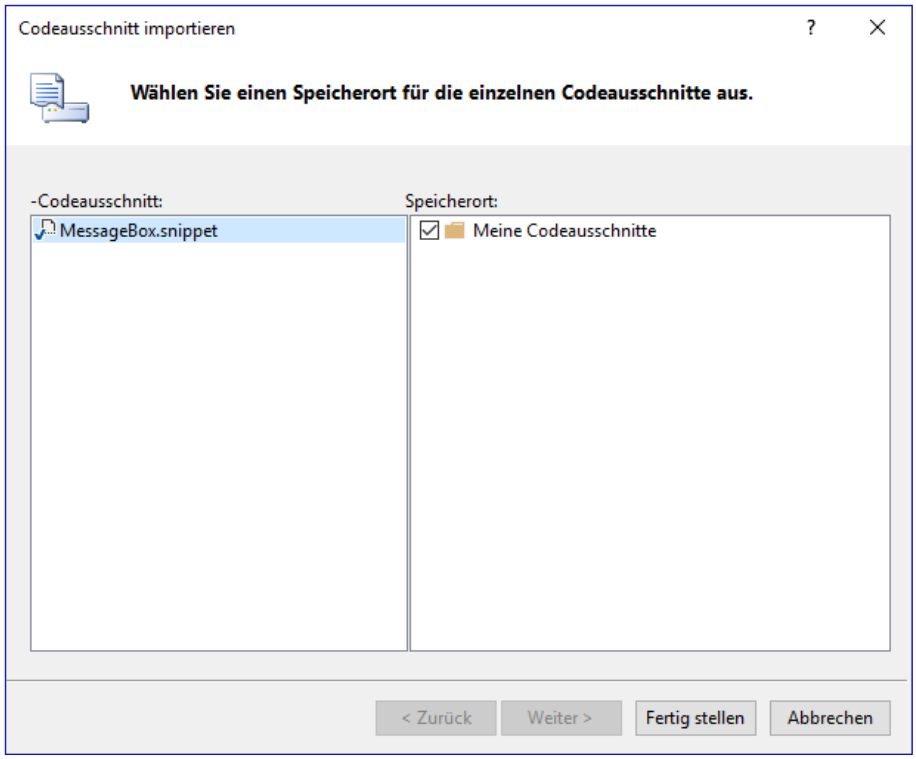


Bild 10: Auswählen des Speicherorts für den Codeschnipsel

Onlinebanking mit DDBAC: Testdaten

Wenn Sie auf Basis der DDBAC-Bibliothek und der hier vorgestellten Techniken eine eigene Onlinebanking-Lösung programmieren, wollen Sie vermutlich nicht ständig Testüberweisungen auf Ihren privaten oder geschäftlichen Konten ausführen, um die Funktionalität zu prüfen. In diesem Fall gibt es gute Nachrichten: Die DDBAC stellt eine Testbank mit zwei Testkonten zur Verfügung, die Sie für diese Zwecke nutzen können. Allerdings gibt es ein paar Einschränkungen bei der Nutzung. Welche diese sind und wie Sie an die Daten der Testkonten gelangen, erfahren Sie in diesem Artikel.

Die Testbank und die beiden Testkonten können Sie beispielsweise nutzen, indem Sie für die eine Testbank einen Account im **Administrator für Homebanking Kontakte** hinzufügen, den Sie nach der Installation der DDBAC über die Systemsteuerung öffnen, und die andere als Ziel einer Überweisung angeben.

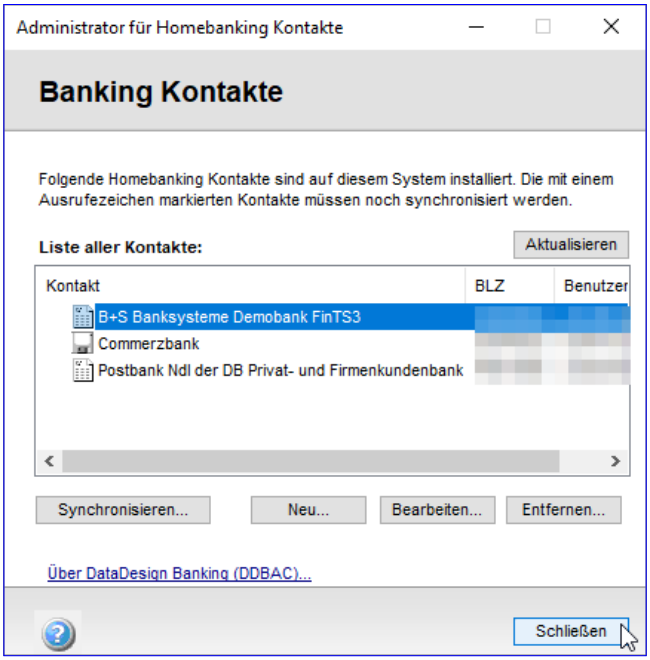


Bild 1: Testzugang im **Administrator für Homebanking Kontakte**

Wenn Sie unter folgendem Link eine DDBAC-Lizenz erworben haben, können Sie per E-Mail an **info@amvshop.de** die Daten der Testkonten anfordern:

```
https://shop.minhorst.com/access-tools/295/ddbac-jahreslizenz?c=78
```

Sie erhalten dann eine Bankleitzahl sowie zwei Kontonummern. Mit der Kombination aus der Bankleitzahl und der ersten Kontonummer legen Sie dann einen neuen Kontakt im Dialog **Banking Kontakte** an. Für das Synchronisieren benötigen Sie auch eine PIN und eine TAN – auch diese erhalten Sie nach Anforderung per E-Mail an **info@amvshop.de**. Nach dem Anlegen sieht der Testzugang im **Administrator für Homebanking Kontakte** wie in Bild 1 aus.

Die Bedingungen für die Nutzung der Testzugänge lauten:

- Sie verwenden nur Verwendungszwecke, die eindeutig als zu Testzwecken erstellt erkennbar sind.
- Sie fügen den Verwendungszwecken ein eindeutiges Element hinzu, damit Ihre Testüberweisungen et cetera identifizieren können, zum Beispiel eine immer gleiche Zahlen- oder Buchstabenkombination.
- Keine Werbung oder strafbaren Inhalte im Verwendungszweck.

Onlinebanking mit DDBAC: FinTS-Registrierung

Wenn Sie Onlinebanking-Lösungen, die Sie mit der DDBAC-Bibliothek realisiert haben, auch in Zukunft ohne Probleme weiterverwenden wollen, müssen Sie zwei zusätzliche Schritte gehen. Diese hängen, wie viele der aktuellen Änderungen im Bereich des Onlinebankings, mit Sicherheitsgründen zusammen. In diesem Fall müssen alle Onlinebanking-Anwendungen bei einer Registrierungsstelle gemeldet werden. Sie erhalten dann eine sogenannte FinTS-Registrierungsnummer, die auch an die Banken verteilt wird. Bei jedem Zugriff über Ihre selbst programmierte Onlinebanking-Anwendung müssen Sie diesen Schlüssel dann an den Bankserver übermitteln, wodurch dieser erkennen kann, welche Anwendung von welchem Hersteller hier zum Einsatz kommt.

Onlinebanking-Anwendung registrieren

Der erste Schritt ist das Programmieren einer eigenen Onlinebanking-Software. Wie das gelingt, erläutern wir in den übrigen Artikeln der Beitragsreihe **Onlinebanking mit DDBAC**.

Wenn Sie eine Onlinebanking-Software eines anderen Herstellers verwenden, wird dieser in der Regel alle notwendigen Schritte erledigen, um diese für die Übermittlung der FinTS-Nummer vorzubereiten. Die Lösung aus der Beitragsreihe **Onlinebanking mit DDBAC** ist noch nicht registriert, da wir die Lösung mit Quellcode liefern.

Die eigene Registrierungsnummer sollte man jedoch nicht weitergeben, da diese gegebenenfalls missbraucht werden kann. Wenn Sie den hier vorgestellten Code also für eine eigene Lösung nutzen wollen, müssen Sie eine eigenen Registrierungsnummer beantragen.

Das ist dann der zweite Schritt, in dem Sie die Registrierungsnummer beantragen. Wie das gelingt, erfahren Sie weiter unten. Nach der Registrierung wird die Nummer auch an die Banken weitergegeben, damit diese Ihre Anwendung an der Registrierungsnummer erkennen können.

Der dritte Schritt ist das Einbauen der Registrierungsnummer so in den Code, sodass diese beim Anmelden an den Bankserver an diesen übergeben wird. Auch diesen Schritt erläutern wir weiter unten im Detail.

Beantragen einer FinTS-Registrierungsnummer

Die FinTS-Registrierungsnummer beantragen Sie unter folgendem Link:

```
https://www.hbci-zka.de/register/prod_register.htm
```

Hier finden Sie ein PDF-Formular, das Sie ausfüllen und an die angegebene E-Mail-Adresse senden. Die Webseite gibt an, dass Sie das Formular, ergänzt um die FinTS-Registrierungsnummer, nach fünf bis zehn Tagen zurückerhalten. Danach kann es allerdings noch wenige Tage dauern, bis diese Registrierungsnummer auch bei allen Banken bekannt ist.

Onlinebanking mit DDBAC II: Überweisungen

Onlinebanking ist aktuell in aller Munde, da es für erhöhte Sicherheit einige Änderungen in den Abläufen gegeben hat. Viele Leser kennen die DDBAC-Bibliothek bereits, denn wir haben in Access im Unternehmen schon darüber berichtet und es ist auch ein Buch zu diesem Thema im André Minhorst Verlag erschienen. Wer diese Bibliothek nutzt, braucht neben einer aktuellen Version auch noch eine kleine Änderung, ohne die Vorgänge nicht mehr durchgeführt werden können. Nachdem wir im ersten Teil der Beitragsreihe das Einlesen von Kontoständen und Umsätzen beschrieben haben, kümmern wir uns nun um das Durchführen von Überweisungen.

Voraussetzungen

Voraussetzung für das Umsetzen der Lösung dieses Artikels ist das Vorhandensein einer DDBAC-Lizenz. Die damit nutzbaren Komponenten sind leider nicht mehr kostenlos verfügbar, sondern müssen lizensiert werden, bevor man diese in seine Produkte einbauen kann. Lizenzen finden Sie im Onlineshop unter **https://shop.minhorst.com/access-tools/295/ddbac-jahreslizenz?c=78**.

Download und Installation

Nach dem Erwerben einer Lizenz erhalten Sie einen Download in ihrem Kundenkonto. Dort finden sie eine Datei namens **DDBACNetWrapper-Version-5-7-65-0.zip** (oder neuer), die Sie entpacken. Darin enthalten ist eine **DDBACSDK.exe**, die Sie auf Ihrem Rechner installieren. Anschließend finden Sie im Verzeichnis **C:\Program Files (x86)\DataDesign\DDBACSDK** die installierten Elemente.

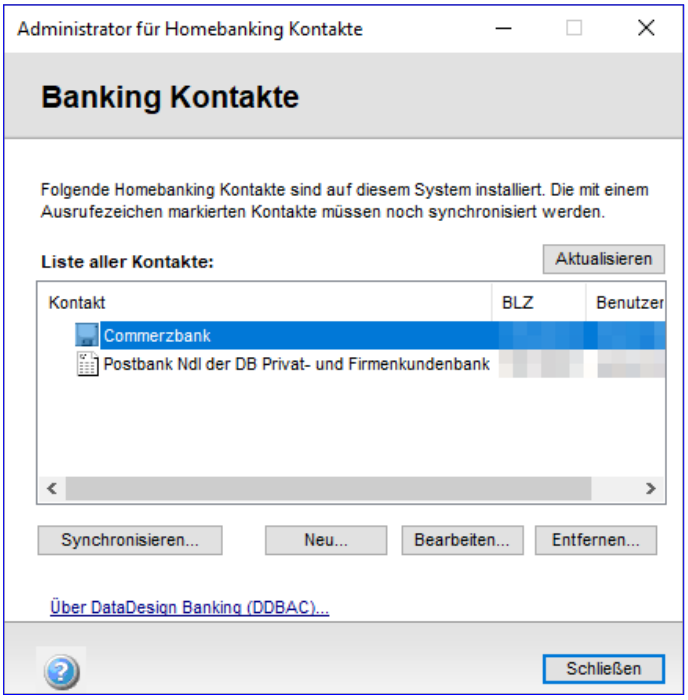


Bild 1: Verwaltung der Homebanking-Kontakte

Test des Quellcodes

Beim Erstellen des Codes für diesen Artikel haben wir das Tool **LinqPad5** zur Programmierung und zum Testen der Funktionalität genutzt. Dieses Tool ist kostenlos. Wir haben dazu im Tool eine neue Datei erstellt und für Language den Wert **VB Program** ausgewählt. Die Methode **Main** wird beim Ausführen aufgerufen, hier haben wir die Aufrufe der einzelnen Beispielmethoden untergebracht.

Überweisung durchführen

Zum Durchführen einer Überweisung verwenden wir diesmal eine **Function**-Methode, da wir einen Wert zurückliefern wollen, der über den Erfolg oder Misserfolg der Überweisung entscheidet. Dieser Funktionswert soll den Datentyp **Boolean** aufweisen. Die benötigten Parameter für eine Überweisung sind die folgenden:

- **lngContact**: Index des Kontakts, für den die Überweisung durchgeführt werden soll (entspricht dem 0-basierten Index der Position des zu wählenden Kontakts aus dem Dialog **Banking Kontakte**, siehe Bild 1)
- **lngAccount**: Index des Kontos des Auftraggebers (entspricht dem 0-basierten Index der Position des zu wählenden Kontos aus dem Dialog Konten verwalten, siehe Bild 2)
- **strEmpfaengername**: Name des Empfängers
- **strEmpfaengerIBAN**: IBAN des Empfänger-Kontos
- **strEmpfaengerBIC**: BIC des Empfänger-Kontos
- **curBetrag**: Zu überweisender Betrag
- **strVerwendungszweck**: Verwendungszweck der Überweisung
- **strMeldung**: Antwort des Systems (mit **ByRef** deklarieren!)

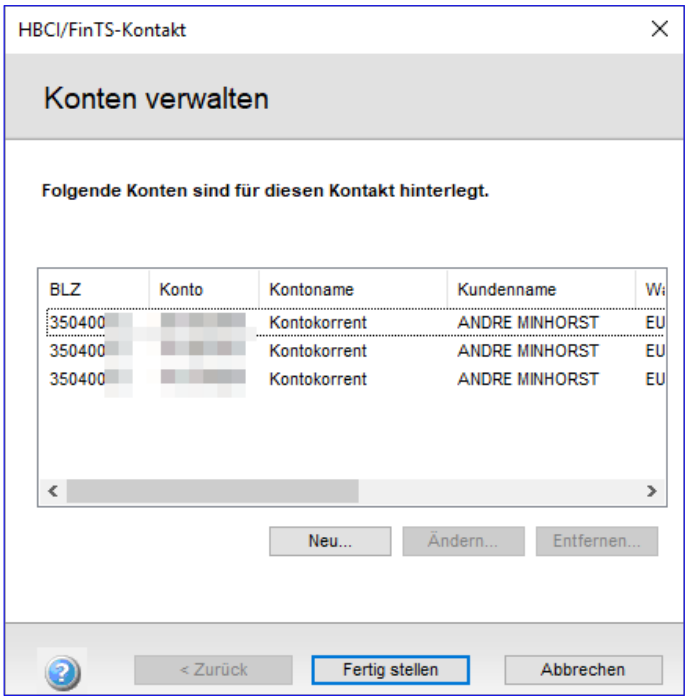


Bild 2: Verwaltung der Konten eines Homebanking-Kontakts

Die nackte Funktion, die wir nachfolgend füllen, sieht nun wie folgt aus:

```
Public Function Ueberweisung(lngContact As Long, lngAccount As Long, strEmpfaengername As String, strEmpfaengerIBAN As String, strEmpfaengerBIC As String, decBetrag As Decimal, strVerwendungszweck As String, ByRef strMeldung As String) As Boolean

End Function
```

BACContact-Objekt für den Auftraggeber ermitteln

Mit dem Parameter **lngContact** wollen wir das **BACContact**-Objekt für den Eintrag im Dialog **Banking Kontakte** auswählen, dessen Index dem Wert dieses Parameters entspricht. Diesen speichern wir in der Variablen **objContact**, die wir wie folgt deklarieren:

```
Dim objContact As BACContact
```

Diese Variable füllen wir dann mit der folgenden Anweisung:

```
objContact = GetContacts.Item(lngContact)
```

Die Funktion **GetContacts** haben wir bereits im ersten Teil der Beitragsreihe namens **Onlinebanking mit DDBAC: Saldo und Umsätze** erläutert. Sie liefert eine Auflistung aller **BACContact**-Elemente, die über die Systemsteuerung hinzugefügt wurden. Über die **Item**-Eigenschaft greifen wir auf das Element mit dem mit **IngContact** übergebenen Indexwert zu.

BACAccount-Objekt für den Auftraggeber ermitteln

Danach benötigen wir das **BACAccount**-Objekt für das Konto, von dem aus die Überweisung durchgeführt werden soll. Dazu legen wir die Variable **objAuftraggeber** an:

```
Dim objAuftraggeber As BACAccount
```

Das **BACAccount**-Objekt ermitteln wir einfach über die **Accounts**-Auflistung des zuvor ermittelten **BACContacts**-Objekts:

```
objAuftraggeber = objContact.Accounts(lngAccount)
```

BACAccount für das Konto des Empfängers erstellen

Während wir das **BACAccount**-Objekt für das Konto des Auftraggebers aus den über die Systemsteuerung abgelegten Daten auslesen konnten, müssen wir das **BACAccount**-Objekt für das Konto des Empfängers neu erstellen und mit den entsprechenden Daten füllen.

Dabei handelt es sich um den Namen, die IBAN und die BIC des Auftraggebers, die wir einem neuen **BACAccount**-Objekt zuweisen. Um diese Daten in einem neuen **BACAccount**-Objekt zusammenzustellen, verwenden wir die folgende Hilfsfunktion **AccountErstellen**:

```
Public Function AccountErstellen(strName1 As String, strIban As String, strBIC As String) As BACAccount
    Dim objAccount As BACAccount
    objAccount = New BACAccount
    With objAccount
        .NameEins = strName1
        .IBAN = strIban
        .BIC = strBIC
    End With
    Return objAccount
End Function
```

Diese erwartet den Namen des Empfängers, die IBAN und die BIC als Parameter und erstellt zunächst ein neues **BACAccount**-Objekt. Dann weist es die Werte der übergebenen Parameter den Eigenschaften **NameEins**, **IBAN** und **BIC** zu. Das so gefüllte **BACAccount**-Objekt wird dann mit der **Return**-Anweisung als Ergebnis der Funktion zurückgegeben.

Die Variable, in der das **BACAccount**-Objekt gespeichert werden soll, heißt **objEmpfaenger** und wird wie folgt deklariert:

```
Dim objEmpfaenger As BACContact
```

Dieses füllen wir über einen Aufruf der zuvor beschriebenen Funktion **AccountErstellen**. Dabei übergeben wir die Werte der drei Parameter **strEmpfaengername**, **strEmpfaengerIBAN** und **strEmpfaengerBIC**:

```
objEmpfaenger = AccountErstellen(strEmpfaengername, strEmpfaengerIBAN, strEmpfaengerBIC)
```

Segmenttyp festlegen und Version ermitteln

Danach speichern wir in einer Variablen namens **strSegment** den Namen des zu verwenden Segments, in diesem Fall **HKCCS**, das der Einreichung einer Einzelüberweisung entspricht:

```
Dim strSegment As String

...
strSegment = "HKCCS"
```

In einer Variablen namens **objSegmentVersion** wollen wir die Version des Segments speichern:

```
Dim objSegmentVersion As BACSegment
```

Dieses ermitteln wir über die Methode **FindOptimalBPDSegment** des **BACAccount**-Elements **objAuftraggeber**, also für die Auftraggeberbank:

```
objSegmentVersion = objAuftraggeber.FindOptimalBPDSegment(strSegment, 0)
```

Abhängig davon, ob ein passendes Segment gefunden werden konnte, steigen wir dann im **If**-Teil der folgenden Abfrage in die eigentliche Überweisung ein oder geben eine Meldung zurück, dass das Segment nicht gefunden werden konnte:

```
If Not objSegmentVersion Is Nothing Then
    Debug.Print ("Segmentversion: " + objSegmentVersion.Version.ToString())
    Return True
Else
    strMeldung = "Segment '" + strSegment + "' konnte nicht gefunden werden."
    Return False
End If
```

HKCCS-Segment erstellen

Nachdem wir die aktuellste Version des **HKCCS**-Segments ermittelt haben, erstellen wir auf dieser Basis das eigentliche **HKCCS**-Segment. Dafür deklarieren wir zwei Variablen – eine für das **BACBanking**-Objekt und eine für das **BACSegment**-Objekt:

```
Dim objBanking As BACBanking
Dim objSegment As BACSegment
```

Das **BACBanking**-Objekt füllen wir mithilfe der Funktion **GetBanking**, die wir gegenüber der Version aus dem ersten Teil der Artikelserie noch geändert haben – wie, lesen Sie im Artikel **Onlinebanking mit DDBAC: FinTS-Registrierung**:

```
Set objBanking = GetBanking
```

Dann nutzen wir die Methode **NewSegment** des **BACBanking**-Objekts und übergeben dieser den Namen des zu erzeugenden Segments, nämlich **HKCCS**, und die soeben ermittelte Version:

```
Set objSegment = objBanking.NewSegment(strSegment, objSegmentVersion.Version)
```

Erfassen der Auftragsdaten

Die Auftragsdaten verarbeiten wir in einem Objekt des Typs **BACSepaMessage**. Dafür deklarieren wir die folgende Objektvariable namens **objSepaMessage**:

```
Dim objSepaMessage As BACSepaMessage
```

Im Code instanzieren wir diese wie folgt:

```
Set objSepaMessage = New BACSepaMessage
```

Das **BACSepaMessage**-Objekt erstellen wir und füllen dann seine Eigenschaft **Segment** mit dem Auftragssegment aus **objSegment**, das wir vorher auf Basis des zu verwendenden Kontos angelegt haben:

```
objSepaMessage = New BACSepaMessage
objSepaMessage.Segment = objSegment
```

Außerdem legen wir dafür über die Eigenschaft **FromAccount** das Konto fest, von dem aus das Geld überwiesen werden soll. Dieses befindet sich in der Variablen **objAuftraggeber**:

AUSGABE 02/2020

DATENBANK ENTWICKLER

MAGAZIN FÜR DIE DATENBANKENTWICKLUNG MIT VISUAL STUDIO FÜR DESKTOP, WEB UND CO.

TOP-THEMEN:

STEUERELEMENTE	Das Expander-Steuerelement	SEITE 3
STEUERELEMENTE	Das TextBox-Steuerelement	SEITE 6
VISUAL BASIC	AndOr und OrElse	SEITE 31
VISUAL BASIC	Zwischenablage programmieren	SEITE 34
ACCESS ZU EDM	Dateien, INotifyPropertyChanged, Validierung	SEITE 44

André Minhorst Verlag

Das Expander-Steuerelement

Manchmal wird es eng in einem Fenster mit vielen oder großen Steuerelementen. Oder Sie haben Steuerelemente, die nicht unbedingt immer sichtbar sein müssen. Dann ist das Expander-Steuerelement genau das Richtige: Mit diesem können Sie Bereiche im Fenster definieren, die bei Bedarf ausgeklappt werden können. Dieser Artikel zeigt die grundlegenden Techniken des Expander-Steuerelements.

Mit dem Kombinationsfeld haben Sie schon ein Steuerelement kennengelernt, mit dem Sie Platz in Ihren Fenstern sparen können – dort zeigen wir die zur Verfügung stehenden Einträge in einem Auswahlfeld an, das nach der Auswahl wieder verschwindet.

Das **Expander**-Steuerelement arbeitet ganz ähnlich. Hier definieren Sie allerdings weitere Steuerelemente, die durch Aufklappen des **Expander**-Steuerelements angezeigt werden können. Das **Expander**-Steuerelement kann allerdings nur ein einziges untergeordnetes Steuerelement aufnehmen.

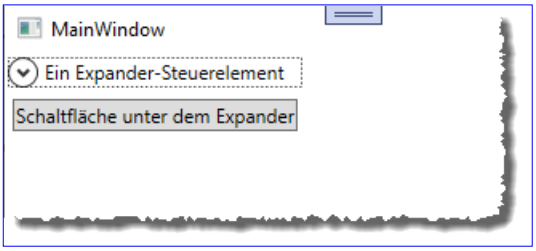


Bild 1: Eingeklapptes **Expander**-Steuerelement

Wenn Sie dennoch mehrere Steuerelemente anzeigen wollen, verwenden Sie eines der verfügbaren Container-Steuerelemente wie beispielsweise das **StackPanel**-Steuerelement. Darin können Sie dann die gewünschten Steuerelemente anlegen. Sie können aber auch ein **Grid**-Steuerelement als Inhalt des **Expander**-Steuerelements angeben.

Ein **Expander**-Steuerelement sieht im eingeklappten Zustand wie in Bild 1 aus. Es zeigt dann nur eine Schaltfläche mit einem Pfeil nach unten an sowie einen Text. Dieser liefert optimalerweise einen Hinweis darauf, welche Steuerelemente sich dahinter verbergen.

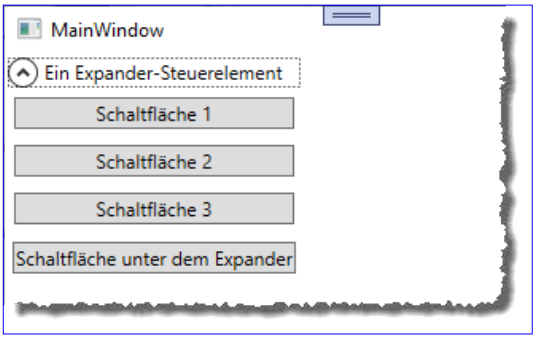


Bild 2: Ausgeklapptes **Expander**-Steuerelement

Nach dem Anklicken des Pfeils nach unten werden die enthaltenen Steuerelemente angezeigt. Die darunter befindlichen Steuerlemente werden dann nach unten verschoben (siehe Bild 2). Außerdem ändert sich das Symbol der Schaltfläche in einen nach oben zeigenden Pfeil.

Der Code hinter diesem **Expander**-Steuerelement sieht wie folgt aus:

```
<Window x:Class="MainWindow" ... Title="MainWindow" Height="450" Width="800">
    <Grid>
        <!--Grid-Definitionen-->
        <Expander Header="Ein Expander-Steuerelement">
            <StackPanel>
```

```
            <Button x:Name="btn1">Schaltfläche 1</Button>
            <Button x:Name="btn2">Schaltfläche 2</Button>
            <Button x:Name="btn3">Schaltfläche 3</Button>
        </StackPanel>
    </Expander>
    <Button Grid.Row="1">Schaltfläche unter dem Expander</Button>
  </Grid>
</Window>
```

Header mit Text oder anderen Inhalten

Statt des hier verwendeten Textes für die Anzeige im Header können Sie auch beliebige anderen Elemente dort anzeigen.

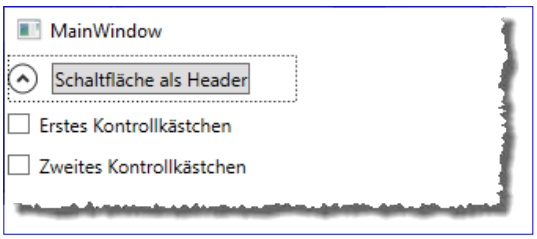


Bild 3: Expander-Element mit Schaltfläche als Header

Sie können dort beispielsweise eine Schaltfläche unterbringen:

```
<Expander Grid.Row="2">
  <Expander.Header>
    <Button>Schaltfläche als Header</Button>
  </Expander.Header>
  <StackPanel Orientation="Vertical">
    <StackPanel Orientation="Horizontal">
      <CheckBox x:Name="chk1"></CheckBox>
      <Label>Erstes Kontrollkästchen</Label>
    </StackPanel>
    <StackPanel Orientation="Horizontal">
      <CheckBox x:Name="chk2"></CheckBox>
      <Label>Zweites Kontrollkästchen</Label>
    </StackPanel>
  </StackPanel>
</Expander>
```

Das Ergebnis finden Sie in Bild 3.

Ausklapp-Richtung festlegen

Mit dem Attribut **ExpandDirection** legen Sie fest, in welche Richtung die enthaltenen Inhalte ausgeklappt werden sollen. Der folgende Expander wird nach oben ausgeklappt:

```
<Expander Grid.Row="3" Header="Ein Expander-Steue-
relement" ExpandDirection="Up">
```

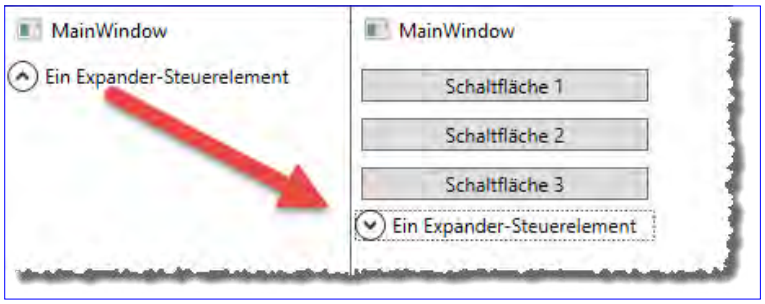


Bild 4: Expander-Inhalt nach oben ausklappen

Das TextBox-Steuerelement

Das wohl am meisten verwendete Steuerelement ist das TextBox-Steuerelement. Es dient zur Eingabe von Texten und bietet eine Menge von Eigenschaften, Ereignissen und Methoden, um damit zu arbeiten. Sie können ungebundene Textfelder nutzen, um Text einzugeben, der auf bestimmte Weise verarbeitet werden soll oder Textfelder über das Entity Data Model an die Datenquelle binden, um Datenbankinhalte anzuzeigen und zu bearbeiten. Dieser Artikel liefert die Grundlagen zum TextBox-Steuerelement unter WPF.

Ein Textfeld ist unter WPF schnell erstellt. Sie fügen einfach das öffnende und das schließende **TextBox**-Element in das Grid-Element ein und fügen dazwischen einen Text hinzu:

```
<Grid>
    <TextBox>Dies ist ein Beispieltext.</TextBox>
</Grid>
```

Das Textfeld nimmt dann allerdings den gesamten Platz im Grid ein, sodass wir noch ein paar Anpassungen vornehmen. Entweder wir legen die Position und die Größe absolut mit den Eigenschaften **Width**, **Height**, **Top** und **Left** fest oder wir fügen zum Grid einige Zeilen und Spalten hinzu, damit wir gleich mehrere Steuerelemente gleichmäßig anordnen können. Wir wählen letztere Alternative, die Sie im Quellcode ansehen können und die im Entwurf wie in Bild 1 aussieht. Hier haben wir noch ein **Label**-Element in die erste Spalte der ersten Zeile eingefügt und das **TextBox**-Element in die zweite Spalte der ersten Zeile:

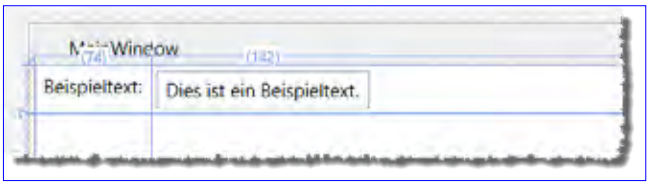


Bild 1: Textfeld mit Bezeichnungsfeld

```
<Label>Beispieltext:</Label>
<TextBox Grid.Column="1">Dies ist ein Beispieltext.</TextBox>
```

Formatierter Text?

Unterschiedliche Textformatierungen lassen sich mit dem **TextBox**-Steuerelement nicht anwenden oder anzeigen. Dazu benötigen wir beispielsweise das **RichTextBox**-Steuerelement.

Padding: Abstand des Inhalts zum Textfeldrand

Neben den Standardeigenschaften **Width**, **Height**, **Top** und **Left** benötigen Sie vermutlich noch die Eigenschaften **Margin** und **Padding**. Mit **Padding** stellen Sie den Abstand des Inhalts, also in der Regel des Textes, von den Begrenzungen des Steuerelements ein.

Soll der Abstand oben, unten, rechts und links gleich groß sein, legen Sie es zum Beispiel mit **Padding="3"** fest. Wenn Sie unterschiedliche Abstände wünschen, geben Sie die Werte durch Kommata getrennt an, wobei die Reihenfolge links, oben, rechts und unten lautet.

Margin: Abstand zu den umgebenden Steuerelementen

Mit der Eigenschaft **Margin** hingegen geben Sie den Abstand an, den das Steuerelement zu den umgebenden Steuerelementen aufweisen soll. Wenn Sie also etwa ein Grid verwenden, um die Steuerelemente anzuordnen, und keinen Wert für **Margin** angeben, gibt es keinen Abstand zwischen den Steuerelementen. Um einen solchen Abstand herzustellen, legen Sie also etwa **Margin="3"** fest – so hat das Steuerelement jeweils einen Abstand von **3** zum nächsten angrenzenden Element, in diesem Fall zu den gedachten Linien des **Grid**-Elements.

Textfeld ausrichten

Für Textfelder gibt es die üblichen Möglichkeiten zur Ausrichtung. Für die horizontale Ausrichtung im Grid verwenden Sie die Eigenschaft **HorizontalAlignment** mit den Werten **Center**, **Left**, **Right** und **Stretch**. Für die vertikale Ausrichtung nutzen Sie die Eigenschaft **VerticalAlignment** mit den Werten **Bottom**, **Center**, **Stretch** und **Top**.

Text ausrichten

Natürlich können Sie auch den Text im Textfeld ausrichten. Dazu verwenden Sie die Eigenschaft **TextAlignment** mit den Werten **Center**, **Justify**, **Left** (Standardwert) und **Right**:

```
<TextBox Text="Mittig zentrierter Text" TextAlignment="Center"></TextBox>
```

Inhalt des Steuerelements

Den Inhalt des Steuerelements können Sie, wie oben gesehen, einfach zwischen das öffnende und schließende **TextBox**-Element einfügen. Dies entspricht der Verwendung der Eigenschaft **Text**:

```
<TextBox Text="Dies ist ein Beispieltext."></TextBox>
```

Eingebautes Kontextmenü

Das **TextBox**-Steuerelement liefert standardmäßig ein Kontextmenü mit den drei Befehlen **Ausschneiden**, **Kopieren** und **Einfügen** mit, das Sie wie üblich mit einem Klick der rechten Maustaste auf das Steuerelement anzeigen (siehe Bild 2). Dieses Kontextmenü können Sie anpassen. Wie das geht, erklären wir in einem weiteren Artikel namens **Kontextmenüs unter WPF** (**www.datenbankentwickler.net/214**).

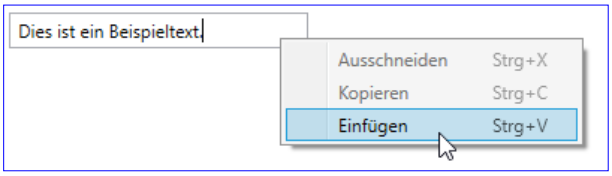


Bild 2: Kontextmenü eines Textfeldes

Weitere Einträge kommen zum Kontextmenü hinzu, wenn die Rechtschreibprüfung aktiviert ist – siehe nächster Abschnitt.

Rechtschreibprüfung

Standardmäßig ist keine Rechtschreibprüfung für **TextBox**-Elemente aktiviert. Sie können diese jedoch leicht anschalten, indem Sie die Eigenschaft **SpellCheck.IsEnabled** auf den Wert **True** einstellen. Wenn Sie dann ein falsch geschriebenes

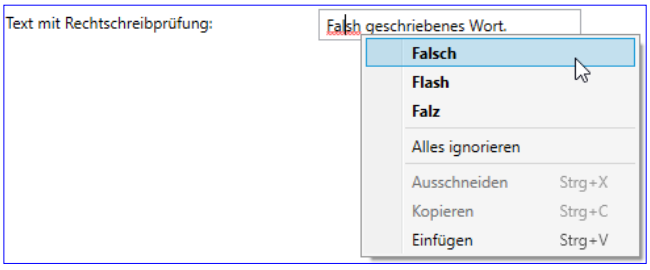


Bild 3: Rechtschreibprüfung im Textfeld mit Verbesserungsvorschlägen

Wort einfügen, wird dieses rot unterstrichen. Klicken Sie dann mit der rechten Maustaste auf das Wort, zeigt das Kontextmenü Verbesserungsvorschläge an (siehe Bild 3).

Eingabe kontrollieren

Wenn Sie sicherstellen wollen, dass nicht mehr als eine bestimmte Anzahl Zeichen in einem Textfeld landen, können Sie die Anzahl mit der Eigenschaft **MaxLength** begrenzen. Geben Sie einfach die maximale Anzahl als Wert an:

```
<TextBox MaxLength="50"></TextBox>
```

Textumbruch aktivieren

Normalerweise wird Text, der die Breite des Textfeldes übersteigt, einfach nicht komplett angezeigt (Voraussetzung dafür ist, dass sich das Textfeld nicht automatisch verbreitern kann – also wenn entweder eine feste Breite festgelegt oder die Breite des Fensters begrenzt ist). Das gilt auch für Textfelder, deren Höhe für die Darstellung von mehr als einer Zeile ausreicht. Wenn sie möchten, dass der Text am Ende des Textfeldes umgebrochen wird, können Sie die Eigenschaft **TextWrapping** verwenden. Diese bietet drei mögliche Werte an:

- **NoWrap**: Standardwert, hier erfolgt kein Zeilenumbruch.
- **Wrap**: Der Text wird am Ende des Textfeldes umgebrochen. Wenn es keine Möglichkeit durch ein Leerzeichen oder ähnliches Zeichen gibt, den Text am Wortende zu umbrechen, dann wird der Umbruch mitten im Wort eingefügt.
- **WrapWithOverflow**: Der Text wird am Ende des Textfeldes umgebrochen. Wenn es keine Möglichkeit durch ein Leerzeichen oder ähnliches Zeichen gibt, den Text am Wortende zu umbrechen, dann erfolgt für dieses Wort kein Umbruch, sondern es wird abgeschnitten angezeigt.

```
<Label Grid.Row="1">Text mit TextWrapping=Wrap:</Label>
<TextBox Grid.Row="1" Grid.Column="1" Height="80" Width="200" TextWrapping="Wrap">
    Dies ist ein Beispieltext mit automatischem Zeilenumbruch undeinemsehrsehrsehrsehrsehrlangenWort.</TextBox>
<Label Grid.Row="2">Text mit TextWrapping=WrapWithOverflow:</Label>
<TextBox Grid.Row="2" Grid.Column="1" Height="80" Width="200" TextWrapping="WrapWithOverflow">
    Dies ist ein Beispieltext mit automatischem Zeilenumbruch undeinemsehrsehrsehrsehrsehrlangenWort.</TextBox>
```

Beispiele sehen Sie in Bild 4.

Zeilenumbrüche mit der Eingabetaste einfügen

Wenn Sie längere Texte im Textfeld eingeben möchten, wollen Sie gegebenenfalls auch Zeilenumbrüche nutzen, die Sie selbst per Eingabetaste festlegen. Auch dafür gibt es eine Eigenschaft. Sie heißt **AcceptsReturn** und erwartet einen der Werte **True** oder **False** (Standardwert). Die folgende Textbox verarbeitet das Betätigen der Eingabetaste als Zeilenumbruch:

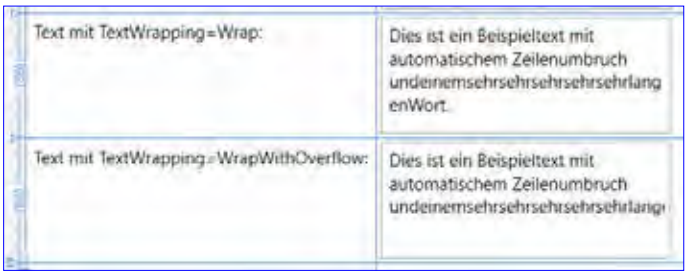


Bild 4: Beispiele für **TextWrapping**-Einstellungen

```
<TextBox Height="80" Width="200" TextWrapping="WrapWithOverflow" AcceptsReturn="True"></TextBox>
```

Damit können Sie beispielsweise Texte wie in Bild 5 erzeugen.

Tabulator aktivieren

Auf ähnliche Weise können Sie die Eingabe von Tabulator-Tastenanschlägen beeinflussen. Normalerweise führt das Betätigen der Tabulator-Taste innerhalb eines Textfeldes dazu, dass der Fokus auf das nächste Steuerelement verschoben wird. Wenn Sie jedoch Tabulator-Zeichen in den Text eingeben möchten, müssen Sie dies aktivieren. Dazu stellen Sie die Eigenschaft **AcceptsTab** auf den Wert **True** ein:

```
<TextBox AcceptsTab="True">
```

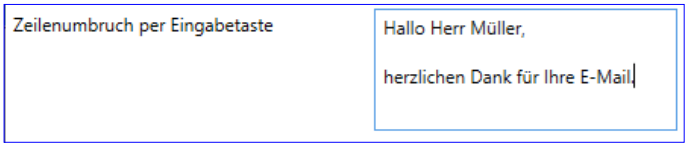


Bild 5: Beispiel für die Eingabe von Zeilenumbrüchen per Eingabetaste

Bildlaufleisten aktivieren

Wenn Sie in einem **TextBox**-Element mehrzeilige Texte anzeigen wollen, kann es sein, dass die Höhe des Textfeldes nicht für die Anzeige des Inhalts ausreicht. In diesem Fall aktivieren Sie die Anzeige der vertikalen Bildlaufleiste, und zwar mit der Eigenschaft **VerticalScrollBarVisibility**:

```
<TextBox Height="80" Width="200" TextWrapping="Wrap" AcceptsReturn="True" VerticalScrollBarVisibility="Auto">Dies ist ein Beispieltext mit automatischem Zeilenumbruch, der zu lang ist, um komplett angezeigt zu werden. Daher haben wir für dieses Textfeld die vertikale Bildlaufleiste aktiviert.</TextBox>
```

Die Eigenschaft bietet die folgenden Werte an:

- **Auto**: Die Bildlaufleiste wird eingeblendet, sobald der Text nicht mehr komplett in das **TextBox**-Element passt.
- **Disabled**: Die Bildlaufleiste wird nicht angezeigt und Sie können auch nicht nach unten scrollen, auch wenn der Text so lang ist, dass ein Teil nicht mehr angezeigt werden kann.
- **Hidden**: Die Bildlaufleiste wird nicht angezeigt (Standardeinstellung). Aber Sie können dennoch durch den Text bis nach unten scrollen.
- **Visible**: Die Bildlaufleiste wird immer angezeigt, auch wenn die Höhe des Textfeldes noch nicht die Höhe des Textfeldes überschreitet.

Bild 6 zeigt ein **TextBox**-Element mit Bildlaufleiste.

Textfeld für die Bearbeitung sperren

Wenn Sie nicht möchten, dass ein Benutzer den Inhalt eines Textfeldes bearbeiten kann (zum Beispiel die ID einer Entität), dann können Sie die Bearbeitung durch Einstellen der Eigenschaft **IsEnabled** auf den Wert **False** sperren:

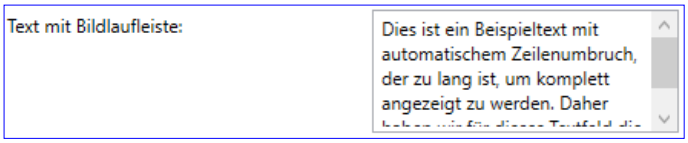


Bild 6: TextBox mit Bildlaufleiste

Kontextmenüs mit WPF

Wer einmal Kontextmenüs unter Access programmiert hat, weiß, wie viel Arbeit das ist. Unter WPF könnte das anders sein. Ob das der Fall ist und wie wir dort Kontextmenüs programmieren, zeigt dieser Artikel. Dabei beleuchten wir, wie Sie ein Kontextmenü für ein bestimmtes Element erzeugen, wie Sie Befehle hinzufügen, diese mit Icons ausstatten, welche Steuerelemente Sie darin unterbringen können und wie Sie die Einträge aktivieren und deaktivieren können. Außerdem erfahren Sie natürlich, wo Sie den Code unterbringen, der durch die Befehle von Kontextmenüs aufgerufen wird.

Das ContextMenu-Element

Kontextmenüs bilden wir unter WPF mit dem **ContextMenu**-Element ab. Dieses können Sie zu beliebigen Elementen hinzufügen – also zum Beispiel zu einem Fenster, einer Seite oder einem dort enthaltenen Steuerelement.

Im einfachsten Fall weisen wir direkt einem **Grid**-Element ein Kontextmenü zu. Der WPF-Code für das **Grid**-Element sieht dann wie folgt aus:

```
<Grid>
    <Grid.ContextMenu>
        <ContextMenu>
            <MenuItem Header="Meldung anzeigen" Click="MenuItem_Click"/>
        </ContextMenu>
    </Grid.ContextMenu>
</Grid>
```

Wer denkt, dass nun beim Rechtsklick auf das **Grid**-Element ein Kontextmenü erscheint, irrt sich: Es geschieht nichts. Und wir haben sichergestellt, dass das Grid sich über das gesamte übergeordnete **Window**-Element erstreckt. Der Clou ist: Jedes Element, das ein Kontextmenü anzeigen soll, muss einen Hintergrund besitzen.

Bild 1: Das erste Kontextmenü, hier für ein **Grid**-Element

Das **Grid**-Element ist schlicht ein Rahmen ohne Inhalt, also können Sie darin auch kein Kontextmenü anzeigen. Das lässt sich allerdings leicht ändern, indem wir dem **Grid**-Element einen Hintergrund zuweisen. Und der darf interessanterweise auch transparent sein. Wir ändern das **Grid**-Element also wie folgt:

```
<Grid Background="Transparent">
```

Danach erscheint beim Rechtsklick das Kontextmenü wie in Bild 1. Zusammengefasst: Wir benötigen ein Property-Element (also eine Eigenschaft, die nicht als Attribut angegeben wird, sondern als untergeordnetes Element) in der Form **<Element-**

typ>.ContextMenu, darin ein **ContextMenu**-Element und darin ein oder mehrere **MenuItem**-Elemente. Mit der Eigenschaft **Header** legen wir den Text des Menüeintrags fest, mit der **Click**-Eigenschaft die Ereignismethode, die beim Anklicken des Eintrags aufgerufen werden soll.

In unserem Fall wollen wir einfach eine Meldung anzeigen und gestalten die Methode wie folgt:

```
Private Sub MenuItem_Click(sender As Object, e As RoutedEventArgs)
    MessageBox.Show("Meldung per Kontextmenü")
End Sub
```

Mehrere Kontextmenü-Einträge

Wollen Sie mehrere Kontextmenü-Einträge anzeigen, fügen Sie einfach die gewünschten Elemente unter dem bereits vorhandenen **MenuItem** ein:

```
<ContextMenu>
    <MenuItem Header="Meldung anzeigen" Click="MenuItem_Click"/>
    <MenuItem Header="Noch ein Eintrag" Click="MenuItem_Click"/>
    <MenuItem Header="Letzter Eintrag" Click="MenuItem_Click"/>
</ContextMenu>
```

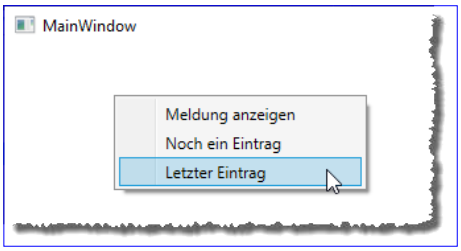


Bild 2: Kontextmenü mit drei Einträgen

Bild 2 zeigt das Ergebnis.

Untermenüs

Wenn Sie Kontextmenü-Einträge unterhalb eines der Einträge benötigen, fügen Sie einfach weitere **MenuItem**-Elemente unterhalb eines der **MenuItem**-Elemente der ersten Ebene ein. Das Ergebnis finden Sie in Bild 3.

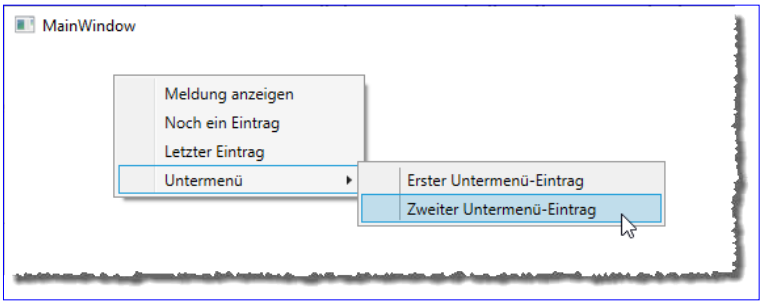


Bild 3: Kontextmenü mit Untermenü-Einträgen

Dem **MenuItem**-Element mit den untergeordneten Elementen können Sie zwar das **Click**-Attribut zuordnen, das Ereignis wird jedoch nicht ausgelöst – der Mausklick bewirkt lediglich das Anzeigen der untergeordneten Elemente:

```
<ContextMenu>
    <MenuItem Header="Meldung anzeigen" Click="MenuItem_Click"/>
    <MenuItem Header="Noch ein Eintrag" Click="MenuItem_Click"/>
    <MenuItem Header="Letzter Eintrag" Click="MenuItem_Click"/>
    <MenuItem Header="Untermenü">
        <MenuItem Header="Erster Untermenü-Eintrag"></MenuItem>
        <MenuItem Header="Zweiter Untermenü-Eintrag"></MenuItem>
    </MenuItem>
</ContextMenu>
```

Click-Ereignisse anlegen

Wenn Sie mehrere Kontextmenü-Einträge in einem Kontextmenü anlegen, haben Sie zwei Möglichkeiten für das Anlegen der dadurch ausgelösten **Click**-Ereignisse: Entweder Sie vergeben jedem Kontextmenü-Eintrag einen eigenen Namen, so wie wir es sonst auch machen. Dann legen sie für jedes Kontextmenü-Element eine eigene Ereignismethode an. Den Namen vergeben Sie wie üblich bei Elementen unter WPF – wir verwenden das Präfix **itm**:

```
<MenuItem x:Name="itmMeldung" Header="Meldung anzeigen" Click="ItmMeldung_Click"/>
```

Die Ereignismethode erhält dementsprechend den Namen **ItmMeldung_Click**. Wenn Sie keine Namen für die Kontextmenü-Einträge vergeben wollen, müssen Sie zumindest Werte mit unterschiedlichen Namen für das **Click**-Attribut festlegen und entsprechend je ein Ereignis für jeden Kontextmenü-Eintrag festlegen.

Trennstriche

Gegebenenfalls möchten Sie zwei Gruppen von Kontextmenü-Einträgen durch einen Trennstrich voneinander abgrenzen. Diesen Trennstrich realisieren wir mit einem Separator-Element (siehe Bild 4):

```
...
<MenuItem Header="Letzter Eintrag" Click="MenuItem_Click"/>
<Separator/>
<MenuItem Header="Untermenü" Click="MenuItem_Click">
...
```

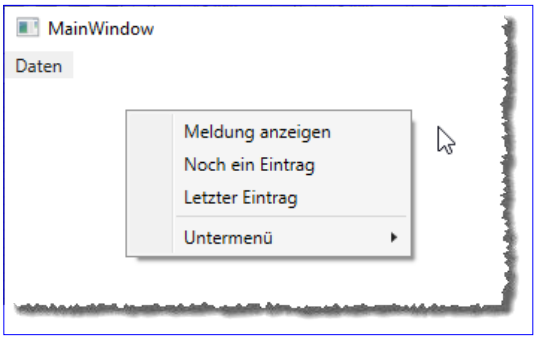


Bild 4: Kontextmenü mit Trennstrich

Kontextmenüs für andere Steuerelemente

Bisher haben wir alle Kontextmenüs direkt dem **Grid**-Element hinzugefügt. Kontextmenüs für andere Steuerelemente fügen Sie auf genau die gleiche Weise hinzu – nur, dass Sie in den meisten Fällen nicht mehr das **Background**-Attribut des jeweiligen Elements anpassen müssen. Hier soll das Kontextmenü beim Rechtsklick auf ein Textfeld angezeigt werden (siehe Bild 5):

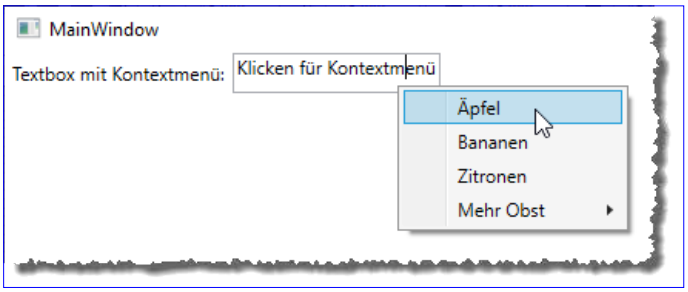


Bild 5: Kontextmenü für ein Textfeld

```
<TextBox Text="Klicken für Kontextmenü">
    <TextBox.ContextMenu>
        <ContextMenu>
            <MenuItem Header="Äpfel"></MenuItem>
            <MenuItem Header="Bananen"></MenuItem>
            <MenuItem Header="Zitronen"></MenuItem>
            <MenuItem Header="Mehr Obst">
                <MenuItem Header="Orange"></MenuItem>
                <MenuItem Header="Ananas"></MenuItem>
            </MenuItem>
```

```
        </ContextMenu>
    </TextBox.ContextMenu>
</TextBox>
```

Icons im Kontextmenü

Bisher sehen die Elemente etwas trist aus. Es wäre doch schön, wenn wir mit passenden Icons ein wenig Farbe und Abwechslung hineinbringen könnten.

Auch dazu gibt es ein passendes Attribut – aber zuerst müssen Sie die Icons, die angezeigt werden sollen, zum Projekt hinzufügen. Wir fügen diese in der Regel in ein Unterverzeichnis namens **Images** ein (siehe Bild 6).

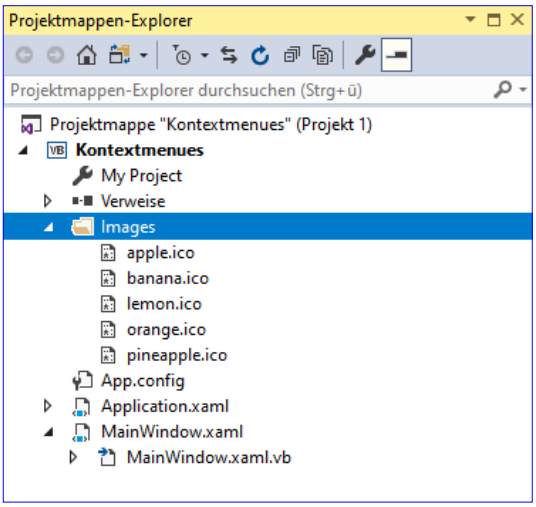


Bild 6: Bilddateien für die Kontextmenü-Einträge

Danach passen wir die Kontextmenü-Einträge an. Dazu benötigen wir wieder ein **Property**-Element, diesmal namens **MenuItem.Image** mit untergeordnetem **Image**-Element.

Dieses legen wir für verschiedene Einträge an und stellen für die **Source**-Eigenschaft den Namen des anzuzeigenden Icons ein. Das Ergebnis sehen Sie in Bild 7:

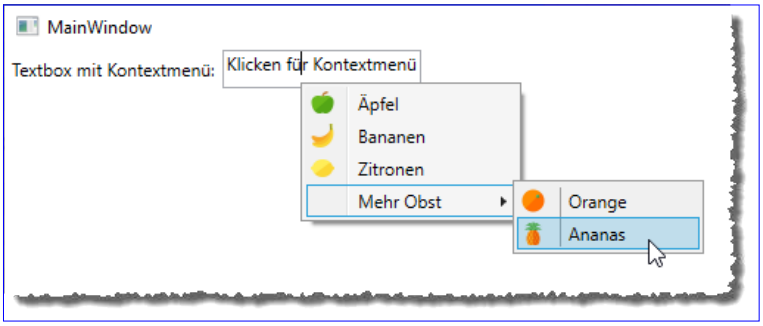


Bild 7: Kontextmenü mit Bildern

```
<TextBox Grid.Column="1" Text="Klicken für Kontextmenü">
    <TextBox.ContextMenu>
        <ContextMenu>
            <MenuItem Header="Äpfel">
                <MenuItem.Icon>
                    <Image Source="Images/apple.ico"></Image>
                </MenuItem.Icon>
            </MenuItem>
            <MenuItem Header="Bananen">
                <MenuItem.Icon>
                    <Image Source="Images/banana.ico"></Image>
                </MenuItem.Icon>
            </MenuItem>
            ...
        </ContextMenu>
    </TextBox.ContextMenu>
</TextBox>
```

Steuerelemente mit Stil

Visual Studio und WPF bieten verschiedene Möglichkeiten, um das Aussehen von Steuerelementen zu beeinflussen. Sie können die Eigenschaften über das Eigenschaftsfenster ändern, die Werte der Attribute direkt für das jeweilige Element im XAML-Code definieren oder auch Stilvorlagen festlegen, die entweder für alle betroffenen Elemente in der Anwendung gültig sind oder auch nur für ein bestimmtes Objekt wie ein Fenster oder eine Seite. Dieser Artikel beschreibt die verschiedenen Möglichkeiten, wie Sie die Steuerelemente Ihrer Anwendung anpassen können.

Eigenschaften per Eigenschaftsfenster

Der offensichtlichste Weg, um Eigenschaften für Steuerelemente einzustellen, ist das Eigenschaftsfenster. Bei einem Button-Element benötigen Sie meist die Eigenschaften der Kategorie Layout (siehe Bild 1). Sie können ein oder mehrere Steuerelemente markieren, deren Eigenschaften Sie einstellen möchten – die eingegebenen Werte werden dann auf alle aktuell markierten Steuerelemente angewendet. Weisen Steuerelemente unterschiedliche Werte für eine der Eigenschaften auf, werden die Eigenschaftswerte nicht angezeigt.

Eigenschaften je Steuerelement per XAML

Die Änderungen der Werte im Eigenschaftsfenster wirken sich unmittelbar auf den XAML-Code des Steuerelements aus. Der XAML-Code enthält nach dem Einfügen eines Steuerelements zunächst einen minimalen Satz von Eigenschaften. Alle Eigenschaften, die nicht als Attribut mit dem Steuerelement aufgeführt werden, enthalten implizit den Standardwert.

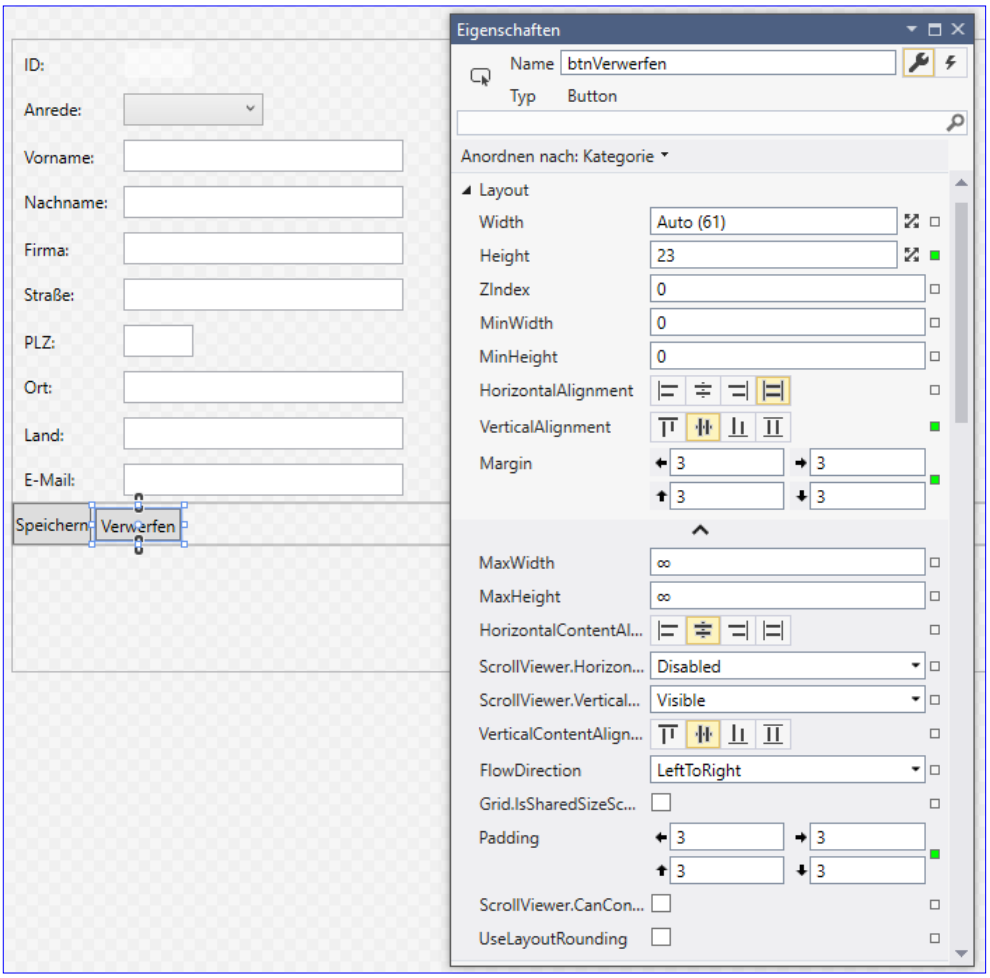


Bild 1: Einstellen per Eigenschaftsfenster

Einheitliche Eigenschaften für mehrere Steuerelemente

Sie können die Attribute natürlich auch direkt im XAML-Code anpassen. Damit erreichen Sie dann umgekehrt auch eine Anpassung der Werte im Eigenschaftsfenster. Bevor Sie ein Attribut für genau ein Steuerelement anpassen, prüfen Sie, ob diese Anpassung wirklich nur für dieses Steuerelement relevant ist. Wenn Sie etwa die **Padding**-Eigenschaft ändern, die den Abstand vom Inhalt zu den Begrenzungen des Steuerelements festlegt, sollten Sie sicher sein, dass diese Eigenschaft wirklich nur explizit für ein Steuerelement geändert werden soll. Alternativ können Sie in Erwägung ziehen, dass es mehrere gleichartige Steuerelemente gibt, für die Sie diese Eigenschaft einheitlich gestalten wollen. Natürlich gibt es Eigenschaften, die tatsächlich nur ein Steuerelement betreffen – zum Beispiel die Beschriftung oder die Namen von Ereignisroutinen, die Sie für die Ereignisattribute angeben. Und auch der Steuerelementname ist natürlich ein individuelles Attribut, das Sie, wenn nötig, für jedes Steuerelement einzeln festlegen.

Das Style-Element

Wenn es jedoch Attribute gibt, die Sie für eine Gruppe von Steuerelementen festlegen wollen, dann können Sie ein mächtiges Feature von WPF nutzen: das **Style**-Element. Der wichtigste Nutzen von **Style**-Elementen ist, Attribute zu definieren, die Sie Steuerelementen zuweisen können, statt die Attribute für jedes Steuerelement einzeln zu definieren. Wenn wir zwei Schaltflächen mit den Attributen **Padding** und **Margin** ausstatten wollen, können wir das für jede Schaltfläche individuell erledigen:

```
<StackPanel Orientation="Horizontal">
    <Button Padding="3" Margin="3">Schaltfläche 1</Button>
    <Button Padding="3" Margin="3">Schaltfläche 2</Button>
</StackPanel>
```

Der direkteste Weg, Attribute für mehrere Steuerelemente zu definieren, ist ein Style, den Sie im übergeordneten Element festlegen. Im folgenden Beispiel haben wir dem **StackPanel**-Element, in dem sich die betroffenen Steuerelemente befinden, ein **StackPanel.Resources**-Element hinzugefügt. Dieses enthält ein **Style**-Element, für das wir den **x:Key** namens **buttonStyle** angegeben haben. Damit kann der Style referenziert werden. Hier haben wir zwei **Setter**-Elemente eingefügt. Das **Setter**-Element gibt mit **Property** das Attribut an und mit **Value** den zu verwendenden Wert. Wichtig bei dieser Variante ist, dass wir dem Attributnamen die **Control**-Klasse voranstellen (**Control.Margin**).

Damit die Attribute auf die **Button**-Elemente angewendet werden, weisen wir diesen den Style explizit mit **Style="{StaticResource buttonStyle}"** zu:

```
<StackPanel Grid.Row="1" Orientation="Horizontal">
    <StackPanel.Resources>
        <Style x:Key="buttonStyle">
            <Setter Property="Control.Margin" Value="3" />
            <Setter Property="Control.Padding" Value="3" />
        </Style>
    </StackPanel.Resources>
    <Button Style="{StaticResource buttonStyle}">Schaltfläche 1</Button>
    <Button Style="{StaticResource buttonStyle}">Schaltfläche 2</Button>
```

```
</StackPanel>
```

Die Schaltflächen sehen nun genauso aus wie die aus dem vorherigen Beispiel. Wenn wir wie in diesem Beispiel nur zwei Schaltflächen mit zwei Attributen ausstatten wollen, scheint dies übertriebener Aufwand zu sein. Allerdings brauchen Sie schon die erste Änderung nur noch an einer Stelle auszuführen. Umso mehr Attribute es gibt und je mehr Steuerelementen Sie diese zuweisen, umso mehr lohnt sich das Anlegen von **Style**-Elementen.

Styles vererben

Wenn Sie beispielsweise zwei verschiedene Typen von Schaltflächen in Ihren Fenstern verwenden wollen, von denen einige die Beschriftung in normaler Breite anzeigen, andere jedoch eine fette Schrift verwenden sollen, müssen Sie keine zwei **Style**-Elemente anlegen, wobei das zweite **Style**-Element nochmal alle Eigenschaften enthält, die das erste bereits definiert hat. Stattdessen gehen wir wie folgt vor und behalten das **Style**-Element namens **buttonStyle** bei. Außerdem legen wir ein weiteres **Style**-Element namens **buttonStyleBold** an, das nur die Eigenschaft **FontWeight** auf den Wert **Bold** festlegt. Damit dieses **Style**-Element auf dem **Style**-Element **buttonStyle** aufbaut, fügen wir diesem das Attribut **BasedOn** hinzu und stellen es auf den Namen des ersten **Style**-Elements ein, also **buttonStyle**. Den beiden Schaltflächen weisen wir dann einmal **buttonStyle** und einmal **buttonStyleBold** zu:

```
<StackPanel Grid.Row="2" Orientation="Horizontal">
    <StackPanel.Resources>
        <Style x:Key="buttonStyle">
            <Setter Property="Control.Margin" Value="3" />
            <Setter Property="Control.Padding" Value="3" />
        </Style>
        <Style x:Key="buttonStyleBold" BasedOn="{StaticResource buttonStyle}">
            <Setter Property="Control.FontWeight" Value="Bold" />
        </Style>
    </StackPanel.Resources>
    <Button Style="{StaticResource buttonStyle}">Schaltfläche 1</Button>
    <Button Style="{StaticResource buttonStyleBold}">Schaltfläche 2</Button>
</StackPanel>
```

Das Ergebnis sehen Sie in Bild 2. Auf diese Weise können Sie schnell den gewünschten Style für ein Steuerelement festlegen und Änderungen der Eigenschaften in den aufeinander aufbauenden **Style**-Elementen sind nach wie vor nur an einer Stelle nötig.

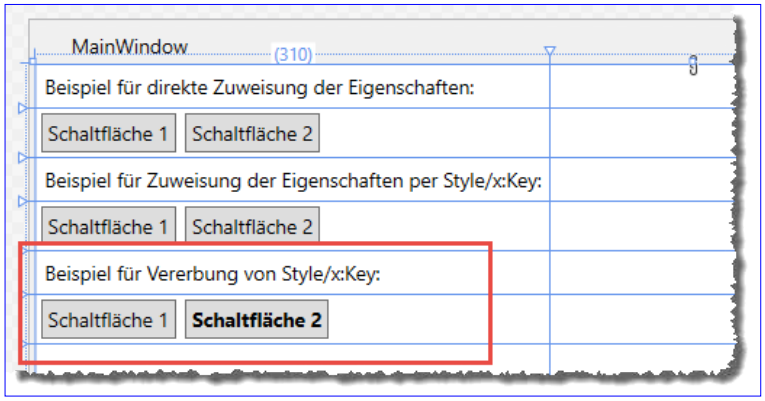


Bild 2: Vererbung eines Stils

Style für bestimmte Objekttypen

Nun haben wir den Style zwar mit **buttonStyle** benannt, aber wir können diesen auch anderen

Visual Basic: AndOr und OrElse

Es gibt oft verschachtelte Bedingungen unter VB. Wer mit VBA programmiert hat, kennt den einzigen Weg, dies zu implementieren: indem man zwei oder mehr If...Then-Bedingungen verschachtelt. Unter Visual Basic nutzen Sie dazu die AndAlso-Verkettung. Und auch bei Ausdrücken mit Oder können Sie mit einem neuen Schlüsselwort die Performance verbessern.

Wenn Sie unter VBA eine Aktion nur durchführen wollen, wenn zwei Bedingungen wahr sind, dann gelingt das am einfachsten mit der folgenden Schreibweise:

```
If Bedingung1 = True And Bedingung2 = True Then
    'Aktion ausführen
End If
```

Zunächst einmal lässt sich dies noch vereinfachen, denn es reicht, die Bedingungen anzugeben – diese werden ja schon auf den Wert **True** geprüft:

```
If Bedingung1 And Bedingung2 Then
    'Aktion ausführen
End If
```

Nur, wenn Sie prüfen wollen, ob der Ausdruck **Bedingung1** oder **Bedingung2** falsch ist, müssen Sie zum Beispiel **Bedingung 1 = False** angeben. Manchmal kommen Sie mit dieser Schreibweise jedoch nicht weiter – zum Beispiel, wenn **Bedingung2** nur auswertbar ist, wenn **Bedingung1** wahr ist. Wenn Sie also in der ersten Bedingung testen, ob ein bestimmtes XML-Element vorhanden ist und in der zweiten dann prüfen, ob der Wert des XML-Elements einen bestimmten Wert hat, funktionierte das nur fehlerfrei, wenn die erste Bedingung erfüllt ist. Ist das nicht der Fall, ist XML-Element nicht vorhanden und die zweite Bedingung löst beim Zugriff auf das XML-Element einen Fehler aus. In diesem Fall verwenden wir zwei verschachtelte **If...Then**-Bedingungen:

```
If Bedingung1 = True Then
    If Bedingung2 = True Then
        'Aktion ausführen
    End If
End If
```

Unter VB können Sie dies viel einfacher formulieren. Dazu nutzen wir das Schlüsselwort **AndAlso**:

```
If Bedingung1 = True AndAlso Bedingung2 = True Then
    'Aktion ausführen
End If
```

Das **AndAlso**-Schlüsselwort bewirkt, dass die zweite Bedingung nur dann ausgewertet wird, wenn die erste Bedingung wahr ist. Allein aus Performance-Gründen sollte man **AndAlso** also dem **And**-Operator vorziehen. Außerdem können Sie auch die Boolean-Ausdrücke an anderen Stellen mit dem AndAlso-Operator verknüpfen – also zum Beispiel in der Abbruchbedingung von Schleifen:

```
Do While Bedingung1 = False AndAlso Bedingung2 = False
...
```

Or oder OrElse?

Neben der Variante **AndAlso** für das **And**-Schlüsselwort gibt es auch noch den **OrElse**-Operator, den Sie alternativ zum **Or**-Operator einsetzen können. Bei einer **Or**-Verknüpfung wie der folgenden werden immer alle Ausdrücke ausgewertet:

```
If Bedingung1 Or Bedingung2 Then
    'Aktion ausführen
End If
```

Wenn Sie hingegen das Schlüsselwort **OrElse** verwenden, wird zunächst **Bedingung1** ausgewertet und nur, wenn diese nicht wahr ist, wird **Bedingung2** herangezogen:

```
If Bedingung1 OrElse Bedingung2 Then
    'Aktion ausführen
End If
```

Testen, ob das stimmt

Wenn Sie sicherstellen wollen, ob **AndAlso** und **OrElse** wie angegeben funktionieren, können Sie als Bedingungen einfach zwei Funktionen verwenden, die beim Aufruf eine Meldung ausgeben. So sehen Sie, ob die zweite Bedingung abhängig von der ersten Bedingung ausgewertet wird.

Die beiden Funktionen definieren wir wie folgt:

```
Public Function Bedingung1 As Boolean
    Debug.Print("Aufruf Bedingung1")
    Return False
End Function

Public Function Bedingung2 As Boolean
    Debug.Print("Aufruf Bedingung2")
    Return True
End Function
```

Die folgende Methode verknüpft die beiden Bedingungen mit dem **And**-Schlüsselwort:

Programmieren der Zwischenablage mit VB

Unter VBA war das Zugreifen auf die Inhalte der Zwischenablage mit dem Einsatz einiger API-Funktionen verbunden. Unter VB und .NET gelingt das wesentlich einfacher. Dieser Artikel zeigt, wie Sie Inhalt in die Zwischenablage kopieren, den Inhalt der Zwischenablage auslesen und weitere Funktionen nutzen wie etwa das Ermitteln des Typs des Inhalts der Zwischenablage.

Die ClipBoard-Klasse

Die **ClipBoard**-Klasse ist das Herzstück der nachfolgend vorgestellten Techniken. Da die Zwischenablage computerweit genutzt wird, also auch von anderen Programmen und auch von Windows selbst, referenzieren wir die **ClipBoard**-Klasse über **My.Computer**. **My** ist eine Funktion, die einen einfachen Zugriff auf verschiedene Klassen bietet, die sich auf den Computer, die Anwendung, Einstellungen, Ressourcen und so weiter beziehen.

Wir referenzieren über **My** die **Computer**-Klasse, die uns auch die **ClipBoard**-Klasse zur Verfügung stellt. Die **ClipBoard**-Klasse bietet wiederum die folgenden Möglichkeiten – in alphabetischer Reihenfolge:

- **Clear**: Leert den Inhalt der Zwischenablage. Vorsicht, da die Zwischenablage ja auch von anderen Anwendungen genutzt wird!
- **ContainsAudio**: Gibt an, ob die Zwischenablage **WaveAudio**-Inhalt enthält.
- **ContainsData(<String>)**: Gibt an, ob die Zwischenablage Daten des mit dem Parameter **String** angegebenen Typs enthält. Beispielwerte für den Parameter sind **Bitmap**, **StringFormat**, **WaveAudio** et cetera.
- **ContainsFileDropList**: Gibt an, ob die Zwischenablage Daten im **FileDrop**-Format enthält.
- **ContainsImage**: Gibt an, ob sich in der Zwischenablage Daten im Bitmap-Format befinden.
- **ContainsText**: Gibt an, ob die Zwischenablage Text enthält.
- **ContainsText(Optional <TextDataFormat>)**: Überladung von **ContainsText**. Gibt an, ob die Zwischenablage Text enthält. Der optionale Parameter ermöglicht eine genauere Prüfung des Inhalts. Hier gibt es die folgenden Werte: **Text** (**0**), **UnicodeText** (**1**), **Rtf** (**2**), **Html** (**3**), **CommaSeparatedValue** (**4**)
- **GetAudioStream**: Liefert einen in der Zwischenablage gespeicherten Audiostream.
- **GetData(<String>)**: Liefert den Inhalt der Zwischenablage in dem mit dem Parameter **String** angegebenen Format.
- **GetDataObject**: Ruft die Daten der Zwischenablage ab – unabhängig vom Typ.

- **GetFileDropList**: Ruft einen oder mehrere Dateinamen aus der Zwischenablage ab.
- **GetImage**: Ruft ein Bild aus der Zwischenablage ab.
- **GetText**: Ruft Text aus der Zwischenablage ab.
- **GetText(<TextDataFormat>)**: Überladung von **GetText**. Ruft Text im angegebenen Textformat ab. Hier gibt es die folgenden Werte: **Text** (**0**), **UnicodeText** (**1**) , **Rtf** (**2**), **Html** (**3**), **CommaSeparatedValue** (**4**)
- **SetAudio(<Byte[]>)**: Fügt ein Byte-Array im WaveAudio-Format als Stream hinzu.
- **SetAudio(<Stream>)**: Überladung von **SetAudio**. Fügt den mit dem Parameter referenzierten Stream im WaveAudio-Format hinzu.
- **SetData(<String>, <Object>)**: Fügt die Daten aus dem Parameter **Object** im mit dem Parameter **String** angegebenen Format in die Zwischenablage ein.
- **SetDataObject(<Object>)**: Fügt ein Objekt in die Zwischenablage ein. Dieses Objekt ist nur temporär bis zum Beenden der auslösenden Anwendung in der Zwischenablage verfügbar.
- **SetDataObject(<Object>, <Boolean>)**: Überladung von **SetDataObject**. Der Parameter **Boolean** gibt an, ob die in die Zwischenablage eingefügten Daten auch nach dem Beenden der Anwendung noch in der Zwischenablage verfügbar sein sollen.
- **SetDataObject(<Object>, <Boolean>, <Int32>, <Int32>)**: Überladung von **SetDataObject**. Die beiden **Int32**-Parameter geben an, wie oft versucht wird, **<Object>** in die Zwischenablage zu übertragen und in welchem zeitlichen Abstand.
- **SetFileDropList(StringCollection)**: Fügt eine Liste von einem oder mehreren Dateipfaden in einer **StringCollection** zur Zwischenablage hinzu.
- **SetImage(<Image>)**: Fügt ein Objekt des Datentyps **Bitmap** zur Zwischenablage hinzu.
- **SetText(<String>)**: Fügt einen Text zur Zwischenablage hinzu.
- **SetText(<String>, <TextDataFormat>)**: Überladung von **SetText**. Fügt einen Text mit dem in **TextDataFormat** angegebenen Format zur Zwischenablage hinzu.

Text in der Zwischenablage

Das Hinzufügen von Text in die Zwischenablage ist schnell erledigt. In einem Beispielprojekt haben wir dazu ein Textfeld namens **txtText** und eine Schaltfläche namens **btnInZwischenablage** hinzugefügt:

```
<Button x:Name="btnInZwischenablage" Click="BtnInZwischenablage_Click">In Zwischenablage</Button>
```

```
<Label>Text:</Label>
<TextBox x:Name="txtZwischenablage" Width="200"></TextBox>
```

Ein Klick auf die Schaltfläche **btnInZwischenablage** soll den aktuellen Inhalt des Textfeldes **txtZwischenablage** in die Zwischenablage kopieren, was wir so realisieren:

```
Private Sub BtnInZwischenablage_Click(sender As Object, e As RoutedEventArgs)
    Clipboard.SetText(txtZwischenablage.Text)
End Sub
```

Das Ergebnis sehen Sie in Bild 1.

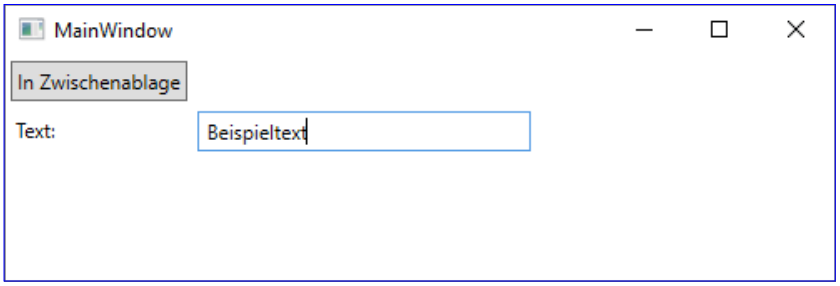


Bild 1: Text aus einem Textfeld in die Zwischenablage kopieren

Text aus der Zwischenablage abrufen

Wenn Sie den soeben in der Zwischenablage gespeicherten Text oder einen anderen in der Zwischenablage gespeicherten Text auslesen wollen, verwenden Sie die Methode **GetText**.

Eine zweite Schaltfläche namens **btnAusZwischenablage** soll den Inhalt der Zwischenablage in das Textfeld kopieren. Dazu verwenden wir schlicht die Methode **GetText** der **ClipBoard**-Klasse:

```
Private Sub BtnAusZwischenablage_Click(sender As Object, e As RoutedEventArgs)
    txtZwischenablage.Text = Clipboard.GetText
End Sub
```

Damit wird allerdings der komplette Inhalt des Textfeldes überschrieben. Um den Inhalt der Zwischenablage an der Position der Einfügemarke einzufügen beziehungsweise anstelle des aktuell markierten Textes, verwenden Sie die folgende Anweisung:

```
txtZwischenablage.SelectedText = Clipboard.GetText()
```

Prüfen, ob die Zwischenablage Text enthält

Wenn die Zwischenablage keinen Text enthält, wird auch nichts in die Zwischenablage eingefügt. Wenn Sie vor dem Einfügen von Text prüfen wollen, ob die Zwischenablage überhaupt Text enthält, können Sie die **ContainsText**-Methode wie folgt verwenden:

```
If Clipboard.ContainsText Then
    txtZwischenablage.SelectedText = Clipboard.GetText()
Else
    MessageBox.Show("Zwischenablage enthält keinen Text.")
```

```
End If
```

Zwischenablage leeren

Mit der **Clear**-Methode der **Clipboard**-Klasse leeren Sie die Zwischenablage:

```
Private Sub BtnZwischenablageLeeren_Click(sender As Object, e As RoutedEventArgs)
    Clipboard.Clear()
End Sub
```

Zwischenablage und Dateien

Wie genau können wir die Zwischenablage mit Dateien nutzen? Wir können beispielsweise eine Liste von Dateien im Windows Explorer markieren, per **Strg + C** in die Zwischenablage kopieren und die Namen dieser Dateien über eine Schaltfläche dann aus der Zwischenablage weiterverarbeiten. Wir fügen wieder zwei Schaltflächen hinzu, welche zum Hinzufügen von Daten zur Zwischenablage und zum Auslesen der Zwischenablage dienen. Außerdem legen wir ein Listenfeld an, in dem wir die Daten aus der Zwischenablage anzeigen wollen:

```
<Button x:Name="btnInZwischenablageDatei" Click="BtnInZwischenablageDatei_Click">In Zwischenablage</Button>
<Button x:Name="btnAusZwischenablageDatei" Click="BtnAusZwischenablageDatei_Click">Aus Zwischenablage</Button>
<Label>Text:</Label>
<ListBox x:Name="lstZwischenablageDatei" Width="300" Height="100"></ListBox>
```

Dann gehen wir davon aus, dass der Benutzer eine oder mehrere Dateien im Windows Explorer markiert und per Strg + C in die Zwischenablage kopiert hat und dann die Schaltfläche **btnAusZwischenablageDatei** betätigt. Dies löst die folgende Methode aus. Für die dort verwendete Variable des Typs **StringCollection** müssen wir der Code behind-Klasse noch einen Verweis auf den folgenden Namespace hinzufügen:

```
Imports System.Collections.Specialized
```

Die **StringCollection** benötigen wir, da die **GetFileDropList** ein Ergebnis dieses Typs zurückliefert. Dementsprechend deklarieren wir eine Variable dieses Typs und noch eine **String**-Variable namens **strDatei**, um die einzelnen Dateien dieser Auflistung zu durchlaufen:

```
Private Sub BtnAusZwischenablageDatei_Click(sender As Object, e As RoutedEventArgs)
    Dim strDateien As StringCollection
    Dim strDatei As String
```

Danach prüft die Prozedur, ob die Zwischenablage überhaupt eine **FileDropList** enthält:

```
    If Clipboard.ContainsFileDropList Then
```

Falls ja, ermitteln wir diese mit **GetFileDropList** und weisen sie der Variablen **strDateien** zu:

Access zu EDM: Dateien erstellen

In vorangegangenen Artikeln haben wir Prozeduren erstellt, mit denen wir das Datenmodell einer Access-Datenbank einlesen und daraus ein Entity Data Model erstellen können. Zusätzlich haben wir auch noch die enthaltenen Daten ausgelesen und Code erzeugt, mit denen eine auf Basis des Entity Data Models erstellte SQL Server-Datenbank gefüllt werden kann. In diesem Artikel wollen wir einen Schritt weitergehen: Bisher haben wir den Code in die Zwischenablage kopiert, sodass der Benutzer diesen noch in die entsprechenden Module des Visual Studio-Projekts kopieren musste. Nun wollen wir direkt die passenden Module als Dateien erstellen, die nur noch in das Projekt gezogen werden müssen.

Die Artikel, in denen wir das Erstellen des Entity Data Models auf Basis des Datenmodells einer Access-Datenbank beschrieben haben, heißen **Von Access zu Entity Framework: Datenmodell** (**www.datenbankentwickler.net/148**), **Von Access zu Entity Framework: Daten** (**www.datenbankentwickler.net/149**) und **Von Access zu Entity Framework: Update 1** (**www.datenbankentwickler.net/164**).

Die hier beschriebenen Prozeduren im VBA-Projekt einer Access-Datenbank lesen die Struktur des Datenmodells der Datenbank aus, die migriert werden soll, und schreibt die ermittelten Codezeilen in die Zwischenablage. Von dort aus sollen diese dann in die betroffenen Zielmodule des Visual Studio-Projekts eingefügt werden. Das ist noch etwas unkomfortabel, sodass wir die Lösung upgraden wollen – und zwar so, dass wir im Verzeichnis der Access-Datenbank neue Dateien und Verzeichnisse erhalten, die wir direkt in den Projektmappen-Explorer von Visual Studio ziehen können.

Wie soll die Lösung aussehen?

Der Ausgangspunkt ist wieder, dass Sie in dem Projekt mit dem zu erstellenden Entity Data Model ein neues Element des Typs **ADO.NET Entity Data Model** hinzufügen (siehe Bild 1). Hier geben Sie direkt die Bezeichnung der Context-Klasse an, die wir später beim Zugriff über das Entity Data Model auf die in der Datenbank gespeicherten Daten verwenden werden – in diesem Fall **RechnungsverwaltungContext**.

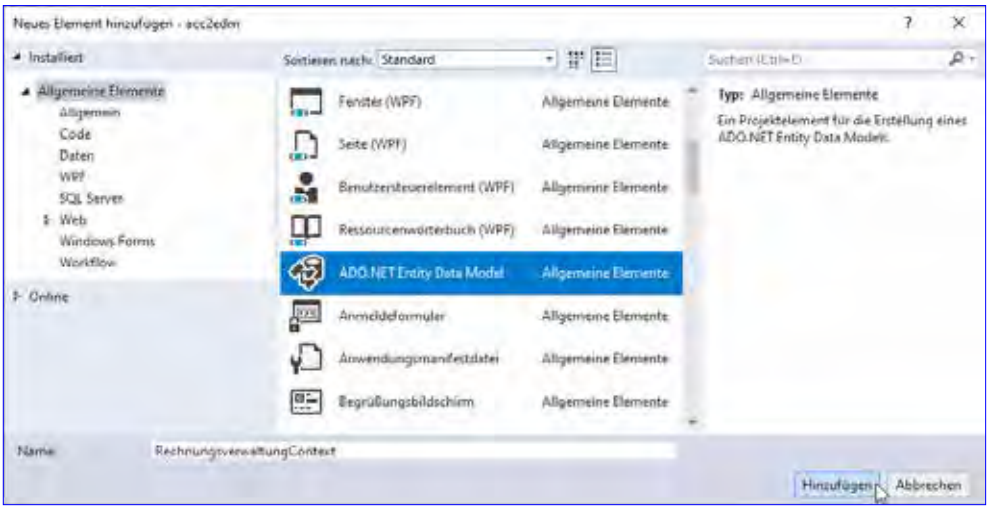


Bild 1: Neues **ADO.NET Entity Data Model**-Element hinzufügen

Im nächsten Schritt wählen wir dann den Eintrag **Leeres Code First-Modell** aus (siehe Bild 2). Damit erscheint im Projektmappen-Explorer zunächst ein neues Element

mit dem Namen, den wir dem neuen **ADO.NET Entity Data Model**-Element zugewiesen haben.

Für die Erstellung des Entity Data Models auf Basis des Datenmodells einer Access-Datenbank merken wir uns diese Bezeichnung.

Die Erstellung erfolgt dann durch einen einmaligen Aufruf einer Prozedur namens **EDMDateienErstellen**. Diese erwartet drei Parameter:

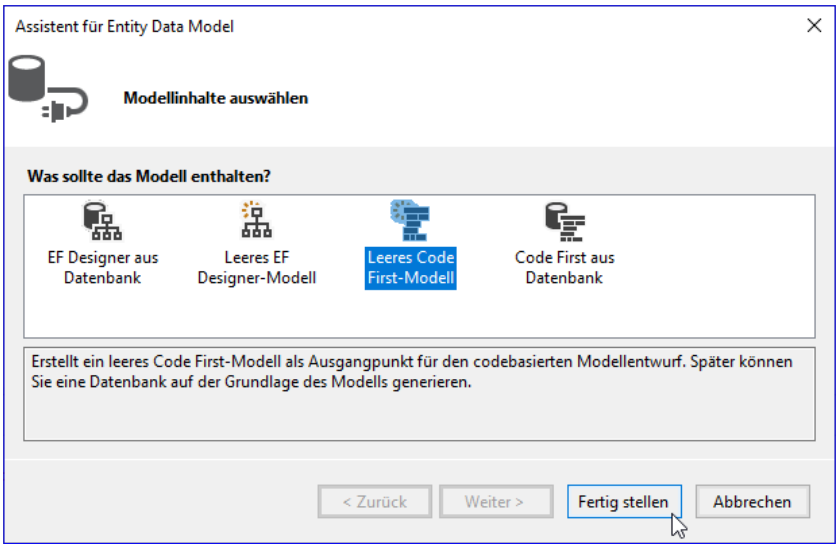


Bild 2: Die Wahl fällt auf **Leeres Code First-Modell**.

- **bolEineDatei**: Gibt an, ob die Entitätsklassen direkt in die Contextklasse geschrieben werden sollen oder als einzelne Dateien in ein Unterverzeichnis.
- **strContextname**: Erwartet den Namen des zuvor erstellten **ADO.NET Entity Data Model**-Elements.
- **strUnterverzeichnisEntities**: Name des Unterverzeichnisses, in das die einzelnen Klassendateien mit den Entitäten geschrieben werden sollen.

Ein Beispielaufruf sieht etwa wie folgt aus:

```
EDMDateienErstellen False, "RechnungsverwaltungContext", "DataModel"
```

Das Ergebnis finden Sie in Bild 3. Die Datei **RechnungsverwaltungContext.vb** enthält die Anweisungen, die Sie auch in der vom Assistenten erstellten gleichnamigen Datei vorfinden – erweitert um die Definition der **DbSet**-Auflistungen wie zum Beispiel die folgende:

```
Public Overridable Property Anreden() As DbSet(Of Anrede)
```

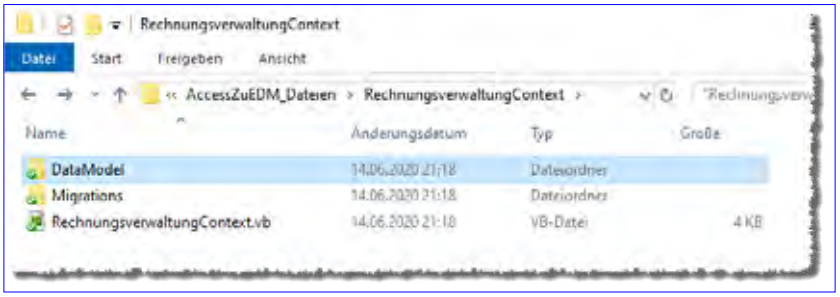


Bild 3: Ergebnis nach dem Aufruf der Prozedur **EDMDateienErstellen**

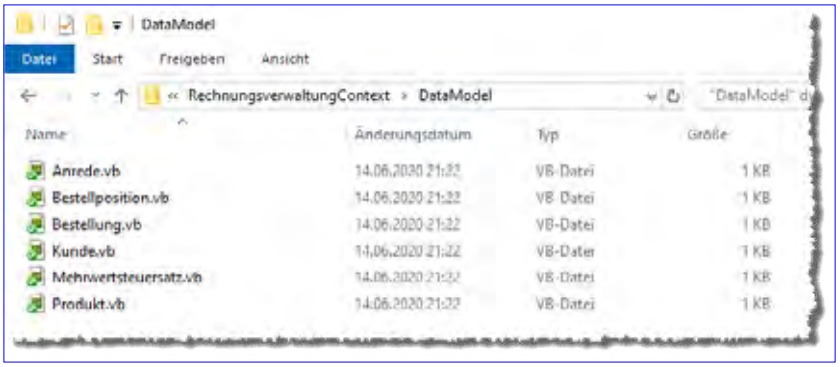


Bild 4: Inhalt des Unterverzeichnisses **DataModel**

Das Verzeichnis **DataModel** enthält die Dateien aus Bild 4. Jede dieser Dateien enthält die Definition einer Entität des Entity Data Models. Die für die Tabelle **tblAnreden** sieht etwa wie folgt aus:

```
Imports System.ComponentModel.DataAnnotations
Imports System.ComponentModel.DataAnnotations.Schema

<Table("Anreden")>
Public Partial Class Anrede
    Public Property ID As System.Int32
    <Index(IsUnique:=true)>
    <StringLength(255)>
    <Required>
    Public Property Name As System.String
End Class
```

Das Ergebnis entspricht also inhaltlich dem aus den in den weiter oben referenzierten Artikeln beschriebenen Prozeduren – mit dem Unterschied, dass diese nun direkt in einem handlichen Verzeichnis liegen. Auf die gleiche Weise werden auch die übrigen Entitätsklassen in jeweils einer Datei in das Verzeichnis kopiert.

Dateien zum Projekt hinzufügen

Nun folgt der wichtigste Schritt: Wir wollen die Dateien, die wir soeben erstellt haben, zum Projekt hinzufügen. Dazu öffnen Sie das neu erstellte Verzeichnis, dass die Unterverzeichnisse **DataModel** und **Migrations** sowie die Datei **RechnungsverwaltungContext.vb** enthält und markieren die enthaltenen Elemente. Diese ziehen Sie dann in den Projektmappen-Explorer auf das Projekt-Element. Der Projektmappen-Explorer sieht danach wie in Bild 5 aus.

Da der Ordner **Migrations** bereits vorhanden ist, brauchen Sie in der Paket-Manager-Konsole nun nur noch zwei der sonst üblichen drei Anweisungen auszuführen, damit die Anweisungen zum Erstellen der Datenbank hinzugefügt und die Datenbank erstellt wird. Die Anweisung **enable-migrations** entfällt beziehungsweise ihr Aufruf liefert nur den Hinweis, dass die Migrationen für dieses Projekt bereits aktiviert wurden.

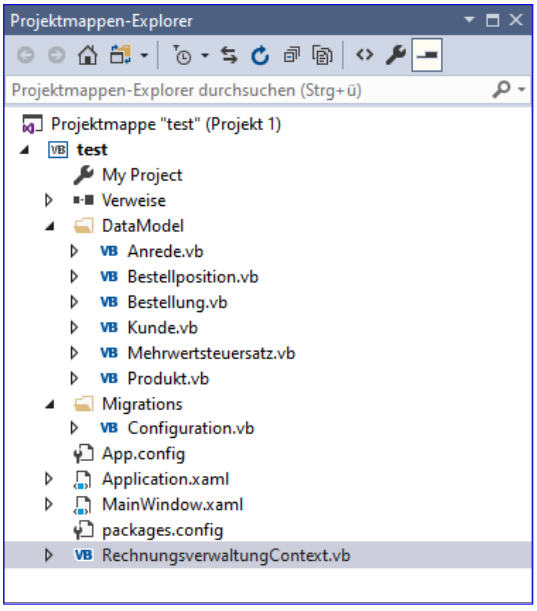


Bild 5: Neue Elemente im Projektmappen-Explorer

Die verbleibenden Anweisungen lauten:

```
add-migration init
update-database
```

Erstere stellt die Methoden zusammen, mit denen die Datenbank gefüllt werden soll, letztere ruft diese auf und erstellt so die Datenbank.

Damit erhalten wir dann die Datenbank aus Bild 6.

Access zu EDM: INotifyPropertyChanged integrieren

Im Beitrag »Access zu EDM: Dateien erstellen« haben wir gezeigt, wie Sie mit einer einfachen Access-Prozedur aus einem Access-Datenmodell die für ein Entity Data Model notwendigen Dateien erzeugen können. Das Entity Data Model enthält nur die reinen Eigenschaften, welche die Felder der jeweiligen Tabellen repräsentieren. Manchmal benötigen Sie allerdings mehr als nur diese Eigenschaften – dann soll zum Beispiel die Schnittstelle INotifyPropertyChanged in der Entitätsklasse implementiert sein, um Änderungen in den Eigenschaften schnell in die Anzeige der Daten übermitteln zu können. Wie Sie die Access-Prozedur zum Erstellen des Entity Data Models entsprechend erweitern, zeigen wir Ihnen in diesem Artikel.

Unsere Entitätsklassen, die wir mit der Prozedur **EDMDateienErstellen** erzeugen, bestehen nur aus den reinen Feldern samt Attributen:

```
<Table("Anreden")>
Public Partial Class Anrede
    Public Property ID As System.Int32
    <Index(IsUnique:=true)>
    <StringLength(255)>
    <Required>
    Public Property Name As System.String
End Class
```

Für die Implementierung der **INotifyPropertyChanged**-Schnittstelle sind einige Erweiterungen notwendig, die wir in den folgenden Abschnitten hinzufügen.

Namespace hinzufügen

Um von der Form der Entitätsklasse, wie wir sie in der bisher verwendeten Prozedur erstellt haben, zu einer Form mit Implementierung der Schnittstelle **IPropertyNotifyChanged** zu gelangen, müssen wir einige Elemente zur Entitätsklasse hinzufügen. Das erste ist der Namespace, in der diese Schnittstelle definiert wird. Dazu passen wir die Funktion **NamespacesFuerEntitaeten** an, indem wir dieser einen Parameter namens **bolINotifyPropertyChanged** hinzufügen. Diesen werten wir in der Funktion aus und fügen, wenn diese den Wert **True** hat, noch die folgende Zeile hinzu:

```
Imports System.ComponentModel
```

In der Funktion sieht das wie folgt aus:

```
Public Function NamespacesFuerEntitaetenZusammenstellen(Optional bolINotifyPropertyChanged As Boolean) As String
    Dim strNamespaces As String
```

```
    If bolINotifyPropertyChanged Then
        strNamespaces = strNamespaces & "Imports System.ComponentModel" & vbCrLf
    End If
    ...
    NamespacesFuerEntitaetenZusammenstellen = strNamespaces
End Function
```

Implements-Anweisung hinzufügen

Als Nächstes wollen wir der Entitätsklasse die **Implements**-Anweisung hinzufügen, mit der wir festlegen, dass die Klasse die Schnittstelle **INotifyPropertyChanged** implementiert. Dieser Teil erfolgt in der Funktion **EntityDataModelZusammenstellen**, die wir von der Hauptprozedur **EDMDateienErstellen** aus aufrufen. Dazu fügen wir dem Aufruf die Übergabe des Parameters **bolINotifyPropertyChanged** hinzu:

```
Public Sub EDMDateienErstellen(bolEineDatei As Boolean, strContextname As String, _
        strUnterverzeichnisEntities As String, Optional bolINotifyPropertyChanged As Boolean)
    ...
    Set colMappings = EntityDataModelZusammenstellen(strDBSets, strEntities, strEntityNames, bolINotifyPropertyChanged)
    ...
End Sub
```

In der Funktion **EntityDataModelZusammenstellen** nehmen wir diesen Parameter im Funktionskopf entgegen und prüfen dann wieder per **If...Then**-Bedingung, ob **bolINotifyPropertyChanged** den Wert **True** hat. In diesem Fall fügen wir der Definition der Klasse in der Variablen **strEntities(i)** die Zeile **Implements INotifyPropertyChanged** hinzu:

```
Public Function EntityDataModelZusammenstellen(strDBSets As String, strEntities() As String, _
        strEntityNames() As String, Optional bolINotifyPropertyChanged As Boolean) As Collection
    ...
    For Each objMapping In colMappings
        With objMapping
            ...
            strEntities(i) = "<Table(""" & .Entities & """)>" & vbCrLf
            strEntities(i) = strEntities(i) & "Public Partial Class " & .Entity & vbCrLf
            If bolINotifyPropertyChanged Then
            If bolINotifyPropertyChanged Then
                strEntities(i) = strEntities(i) & "    Implements INotifyPropertyChanged" & vbCrLf
                strEntities(i) = strEntities(i) & vbCrLf
                strEntities(i) = strEntities(i) & "    Public Event PropertyChanged As PropertyChangedEventHandler " _
                    & "Implements INotifyPropertyChanged.PropertyChanged" & vbCrLf
                strEntities(i) = strEntities(i) & vbCrLf
                strEntities(i) = strEntities(i) & "    Protected Overridable Sub OnPropertyChanged(propname As String)" _
                   & vbCrLf
```

```
                strEntities(i) = strEntities(i) _
                    & "        RaiseEvent PropertyChanged(Me, New PropertyChangedEventArgs(propname))" & vbCrLf
                strEntities(i) = strEntities(i) & "    End Sub" & vbCrLf
                strEntities(i) = strEntities(i) & vbCrLf
            End If
    ...
End Function
```

Wie Sie sehen, fügen wir auf die gleiche Weise auch gleich die Implementierung dieser Schnittstelle hinzu. Dabei handelt es sich um die folgenden Zeilen, die unabhängig vom Klassennamen immer gleich lauten:

```
Public Event PropertyChanged As PropertyChangedEventHandler Implements INotifyPropertyChanged.PropertyChanged

Protected Overridable Sub OnPropertyChanged(propname As String)
    RaiseEvent PropertyChanged(Me, New PropertyChangedEventArgs(propname))
End Sub
```

Eigenschaften um den OnPropertyChanged-Aufruf erweitern

Im letzten Schritt müssen wir noch die als automatisch implementierten Eigenschaften in der langen Form schreiben, damit wir den Aufruf der Methode **OnPropertyChanged** in den Setter einfügen können. Bisher sieht eine Eigenschaft beispielsweise so aus:

```
    <Required>
    Public Property Bezeichnung As System.String
```

Sie soll nun aber so aussehen:

```
    Private _Bezeichnung As String
    Public Property ID As System.Int32
    <StringLength(255)>
    Public Property Bezeichnung As System.String
        Get
            Return _Bezeichnung
        End Get
        Set
            _Bezeichnung = Value
        End Set
        OnPropertyChanged("Bezeichnung")
    End Property
```

Access zu EDM: Validierung

Im Beitrag »Access zu EDM: Dateien erstellen« haben wir gezeigt, wie Sie mit einer einfachen Access-Prozedur aus einem Access-Datenmodell die für ein Entity Data Model notwendigen Dateien erzeugen können. Wenn Sie nun noch passende Klassen mit den Grundfunktionen für die Validierung hinzufügen wollen, können Sie die Erweiterungen aus diesem Artikel dazu verwenden. Wir legen für jede Entitätsklasse des Entity Data Models auf Basis des Datenmodells einer Access-Datenbank zusätzlich eine weitere Klasse in einem Unterordner namens Validierung an, der die grundlegenden Funktionen für die Validierung enthält. Diese müssen allerdings noch angepasst werden. An welchen Stellen, lesen Sie im Folgenden.

Die grundlegenden Techniken zum Validieren von Daten in Benutzeroberflächen, die Daten aus einem Entity Data Model nutzen, haben wir im Artikel **Validieren mit VB und EDM** beschrieben (**www.datenbankentwickler.net/175**). Dort zeigen wir eine Methode, die in verschiedenen Bereichen implementiert werden muss. Der erste dort beschriebene Teil erläutert, wie Sie die Entitätsklassen um eine Schnittstelle erweitern, die das Auslösen von Ereignissen beim Eintreten von Validierungsfehlern erlaubt. Diese Schnittstelle implementieren wir nicht direkt in der Klasse, zum Beispiel **Kunde**, sondern machen uns zunutze, dass man Klassen auf mehrere Dateien aufteilen kann. Das hat den Vorteil, dass automatisch erzeugte neue Versionen des Entity Data Models nach Änderungen im Datenmodell sich nicht auf die Klassen mit der Validierung auswirken.

Wir erstellen also eine neue partielle Klasse, welche die für die Validierung notwendigen Elemente aufnimmt und das ist genau das, was wir automatisch auf Basis des Datenmodells der Access-Datenbank erledigen wollen.

Die übrigen Schritte, die im oben genannten Artikel beschrieben werden, beziehen sich auf die Änderungen, die per XAML für die Benutzeroberfläche umgesetzt werden müssen.

Benötigter Code für die partielle Klasse mit der Validierung

Die partiellen Klassen, welche den Code für die Validierung enthalten, wollen wir wie gesagt in jeweils einer eigenen Klassendatei unterbringen, die in einem Unterverzeichnis namens **Validation** liegen. Eine solche Klasse soll wie folgt aussehen und zunächst den Namespace importieren, der die benötigte Schnittstelle enthält:

```
Imports System.ComponentModel
```

Dann folgt die Klasse, welche die Schnittstelle **IDataErrorInfo** implementiert:

```
Partial Public Class Kunde
    Implements IDataErrorInfo
```

Die Implementierung besteht aus zwei öffentlichen Eigenschaften, von denen uns nur die erste interessiert – die zweite muss aber dennoch implementiert werden. Die erste sieht wie folgt aus:

```
    Default Public ReadOnly Property Item(columnName As String) As String Implements IDataErrorInfo.Item
        Get
            Dim strErrorMessage As String = ""
            Select Case columnName
                Case "Vorname"
                    If (String.IsNullOrEmpty(Vorname)) Then
                        strErrorMessage = "Bitte geben Sie einen Vornamen ein."
                    End If
                ...
            End Select
            Return strErrorMessage
        End Get
    End Property
```

Die Methode erwartet den Namen der zu untersuchenden Eigenschaft als Parameter und unterscheidet diese in einer **Select Case**-Bedingung. Hier prüft sie die Bedingung und trägt dann die Validierungsmeldung in die Variable strErrorMessage ein, die dann auch zurückgegeben wird.

Den Rest der Schnittstelle fügen wir wie folgt an:

```
    Public ReadOnly Property [Error] As String Implements IDataErrorInfo.Error
        Get
            Throw New NotImplementedException()
        End Get
    End Property

End Class
```

Nun benötigen wir eine VBA-Prozedur analog zu der im Artikel **Access zu EDM: Dateien erstellen** (**www.datenbankentwickler.net/219**), die uns diesen Code auf Basis des Datenmodells erstellt. Da wir in den dort vorgestellten Prozeduren und Funktionen schon sehr viele der auch für die Erstellung der Validierungsklassen benötigten Informationen zusammentragen, werden wir diese nochmals erweitern. Das geschieht so, dass die ermittelten Informationen genutzt werden, in weiteren Variablen die Inhalte der Validierungsklassen zusammenzustellen, die wir dann von der Hauptprozedur aus in die Validierungsdateien in einem neuen Verzeichnis namens **Validation** ausgeben.

Die Prozedur **EDMDateienErstellen** erweitern wir dazu zunächst wieder um einen Parameter, diesmal namens **bolValidation**:

```
Public Sub EDMDateienErstellen(bolEineDatei As Boolean, strContextname As String, strUnterverzeichnisEntities _
        As String, Optional bolINotifyPropertyChanged As Boolean, Optional bolValidation As Boolean)
```

Außerdem fügen wir die folgende Variable hinzu:

```
    Dim strValidation() As String
```

Die Variable **strValidation** übergeben wir als neuen Parameter beim Aufruf der Funktion **EntityDataModelZusammenstellen**:

```
    ...
    Set colMappings = EntityDataModelZusammenstellen(strDBSets, strEntities, strEntityNames, strValidation, _
        bolINotifyPropertyChanged, bolValidation)
    ...
```

Die fertigen partiellen Klassen mit dem Validierungscode landen dann in der Array-Variablen **strValidation**.

Weiter unten prüfen wir dann, ob **bolValidation** den Wert **True** hat. In diesem Fall erstellen wir ein neues Unterverzeichnis namens **Validation** und fügen dort in einer Schleife die von der Funktion **EntityDataModelZusammenstellen** gelieferten und im Array **strValidation(i)** gespeicherten Klassen in jeweils eine Datei im soeben erstellten Verzeichnis:

```
    ...
    If bolValidation = True Then
        MakeDirectory strBasispfad & "\Validation"
        For i = LBound(strEntities) To UBound(strEntities)
            TextdateiErstellen strBasispfad & "\Validation\" & strEntityNames(i) & "_Validation.vb", strValidation(i)
        Next i
    End If
End Sub
```

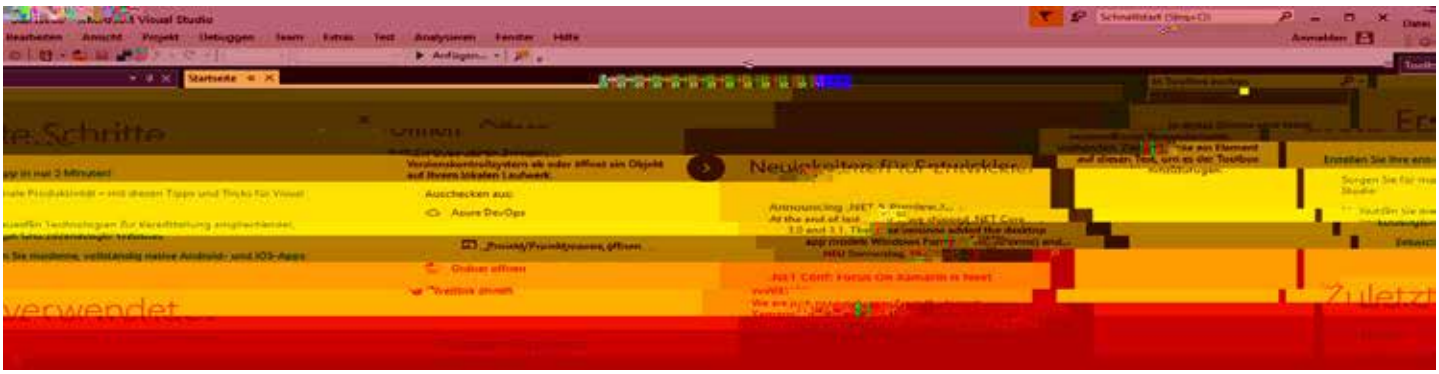
Die größte Arbeit übernimmt wieder die Funktion **EntityDataModelZusammenstellen**. Sie nimmt die neuen Parameter **strValidation** und **bolValidation** entgegen, von denen letztere entscheidet, ob eine Validierung erfolgen soll. Außerdem benötigen wir einige neue Variablen:

```
Public Function EntityDataModelZusammenstellen(strDBSets As String, strEntities() As String, _
```

AUSGABE 03/2020

DATENBANK ENTWICKLER

MAGAZIN FÜR DIE DATENBANKENTWICKLUNG MIT VISUAL STUDIO FÜR DESKTOP, WEB UND CO.



TOP-THEMEN:

LÖSUNGEN	Rechnungsverwaltung, Teil 1: Grundlagen	SEITE 3
LÖSUNGEN	Rechnungsverwaltung, Teil 2: Rechnungspositionen	SEITE 23
REPORTING	Rechnungsbericht mit XAML	SEITE 42
SQL SERVER & CO.	Rechnungsverwaltung auf SQLite umstellen	SEITE 56

André Minhorst Verlag

Rechnungsverwaltung, Teil 1: Grundlagen

Als selbständiger oder freiberuflicher Softwareentwickler braucht man am Ende vor allem eines: eine Anwendung zum Erstellen von Rechnungen. Diese wollen wir im vorliegenden Artikel programmieren – vom Entwurf des Datenmodells über die Erstellung des Entity Data Models und die Benutzeroberfläche bis zum Ausdrucken der Rechnung als PDF oder mit dem Drucker.

Funktionen der Rechnungsverwaltung

Welche Aufgaben wollen wir mit der Rechnungsverwaltung erledigen – welche Eingangsdaten haben wir, was soll am Ende herauskommen? Und welche Funktionen bilden wir dabei mit der Benutzeroberfläche ab?

Wir wollen eine kleine Anwendung bauen, die verschiedene Funktionen über das Ribbon anbietet. Dazu gehören die Verwaltung von Rechnungsempfängern und der Rechnungen selbst. Für die Rechnungsempfänger wollen wir eine Übersicht der vorhandenen Kunden in Listenform anbieten und ein Fenster zum Anlegen eines neuen Kunden. Die Seite zum Anlegen eines Kunden wollen wir auch für die Detailansicht dieses Kunden nutzen. Praktisch ist dann, wenn wir direkt die Rechnungen, die bisher für diesen Kunden erstellt wurden, in einer Liste in dieser Ansicht anzeigen. Außerdem soll das Fenster jeweils eine Schaltfläche zum Anlegen einer neuen Rechnung enthalten und eine zum Ausdrucken einer markierten Rechnung.

Für die Rechnungen wollen wir ebenfalls ein eigenes Fenster vorsehen. Dieses enthält Steuerelemente zur Eingabe der allgemeinen Daten einer Rechnung, also Rechnungsdatum, Rechnungsbetreff, Rechnungstext und Rechnungsuntertext. Außerdem soll ein Listensteuerelement alle Rechnungspositionen mit Bezeichnung, Nettobetrag, Menge und Mehrwertsteuersatz anzeigen.

Schließlich fehlt noch die Ausgabe der Rechnung – diese wollen wir über ein FlowDocument realisieren, das in einem entsprechenden Steuerelement in einem eigenen Fenster angezeigt wird und ausgedruckt oder als PDF gespeichert werden kann.

Datenmodell der Anwendung

Das Datenmodell haben wir, wenn wir schon im Artikel **Access zu EDM: Dateien erstellen** (**www.datenbankentwickler.net/219**) eine kleine Funktion zum Erstellen eines Entity Data Models programmiert haben, schnell mit Access definiert (siehe Bild 1). Wir verwenden eine Tabelle namens **tblKunden**,

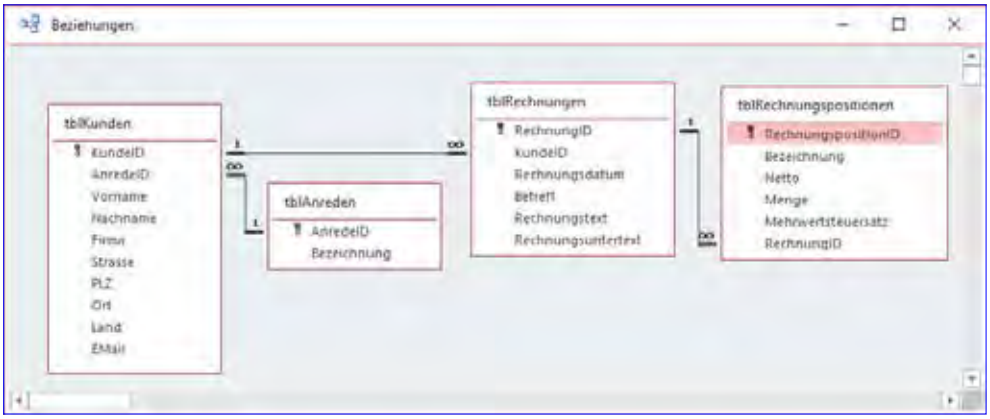


Bild 1: Datenmodell der Rechnungsverwaltung

welche die üblichen Kundendaten enthält. Die Anreden speichern wir in einer eigenen Tabelle namens **tblAnreden**, die von der Tabelle **tblKunden** aus über das Feld **AnredeID** referenziert wird. Die grundlegenden Rechnungsdaten speichern wir in der Tabelle **tblRechnungen**. Hier finden Sie das Rechnungsdatum und die übrigen Informationen, die in der Rechnung rund um die Auflistung der Rechnungspositionen abgebildet werden. Außerdem enthält die Rechnungstabelle ein Fremdschlüsselfeld namens **KundeID**, mit dem der Kunde zur jeweiligen Rechnung festgelegt wird. Schließlich folgen noch die Rechnungspositionen in der Tabelle **tblRechnungspositionen**. Die einzelnen Positionen sind über das Feld **RechnungID** mit der Tabelle **tblRechnungen** verknüpft.

Achtung: Damit das nachfolgende Erstellen des Entity Data Models funktioniert, müssen die Primärschlüsselfelder die Entität im Namen enthalten, bei **Kunde** also beispielsweise **KundeID**.

Entity Data Model erstellen

Mit dem folgenden Aufruf der Funktion **EDMDateienErstellen** legen wir im Unterverzeichnis **RechnungsverwaltungContext** der Access-Datenbank einige Verzeichnisse und Dateien an, die wir gleich im Anschluss weiterverarbeiten:

```
EDMDateienErstellen False, "RechnungsverwaltungContext", "DataModel", True, True
```

Zuvor erstellen wir noch ein neues Visual Studio-Projekt mit der Vorlage **Visual Basic|Windows Desktop|WPF-App** namens **Rechnungsverwaltung**.

Nun öffnen wir mit dem Kontextmenü-Eintrag **Hinzufügen|Neues Element...** des Projekt-Elements im Projektmappen-Explorer den Dialog **Neues Element hinzufügen**, wählen dort das Element **ADO.NET Entity Data Model** aus und geben dafür den Namen **RechnungsverwaltungContext** ein.

Im folgenden Schritt wählen wir den Eintrag **Leeres Code First-Modell** aus (siehe Bild 2).

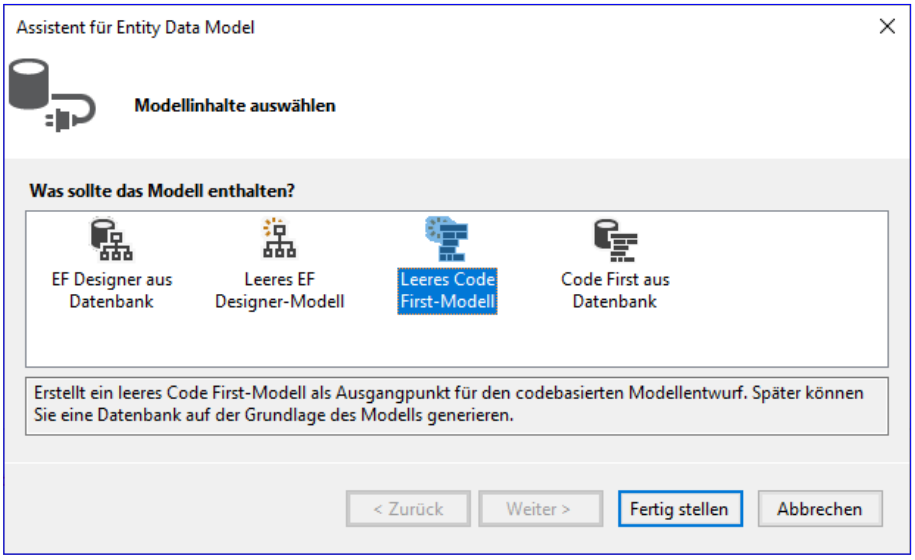


Bild 2: Typ des Modells auswählen

Danach ziehen Sie den Inhalt aus dem soeben von Access aus erstellten Ordner **RechnungsverwaltungContext** (alle enthaltenen Elemente, nicht den Ordner selbst) auf das Projekt-Element **Rechnungsverwaltung**. Dies ersetzt die Datei **RechnungsverwaltungContext.vb**, was sich Visual Studio mit der Meldung aus Bild 3 bestätigen lässt.

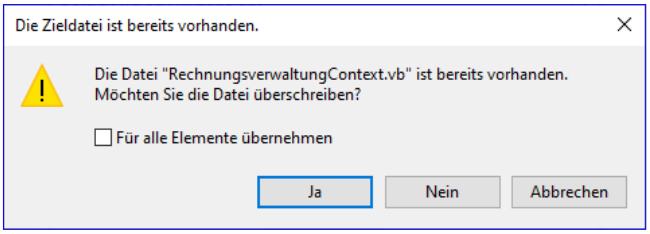


Bild 3: Überschreiben der Datei bestätigen

Mit einer weiteren Meldung bestätigen Sie außerdem, dass die extern geänderten Elemente in Visual Studio neu geladen werden sollen.

Datenbank erstellen

Wenn Sie jetzt noch in der Paket-Manager-Konsole über den Menüeintrag **Extras|Nuget-Paket-Manager|Paket-Manager-Konsole** die folgenden beiden Befehle aufrufen, wird auch noch eine SQL Server-Datenbank auf Basis des Entity Data Models erstellt, die wir im weiteren Verlauf benötigen:

```
add-migration init
update-database
```

Danach finden Sie die Datenbank im SQL Server-Objekt-Explorer vor und können gleichzeitig über das Entity Data Model auf die enthaltenen Daten zugreifen beziehungsweise neue Daten schreiben.

Ribbon der Anwendung

Damit wir die einzelnen Bereiche der Anwendung öffnen können, fügen wir dem Fenster **Main.xaml** ein Ribbon hinzu. Dazu müssen wir zunächst einen Verweis hinzufügen (Menüeintrag **Projekt|Verweis hinzufügen...**), und zwar auf die Bibliothek **System.Windows.Controls.Ribbon** (siehe Bild 4). Danach fügen wir dem Code des Fensters **Main.xaml** die folgende Ribbon-Definition hinzu:

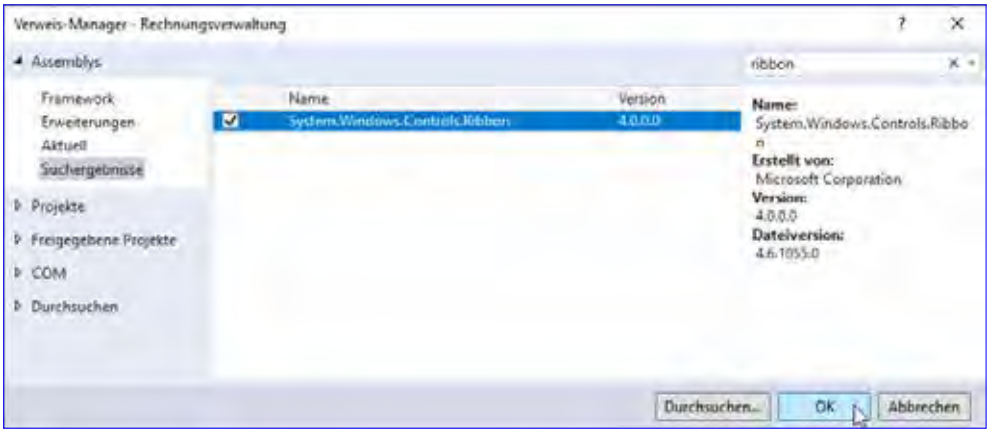


Bild 4: Hinzufügen eines Verweises auf die Bibliothek **System.Windows.Controls.Ribbon**

```
<Grid>
    <Grid.RowDefinitions>
        <RowDefinition Height="Auto"></RowDefinition>
        <RowDefinition Height="*"></RowDefinition>
    </Grid.RowDefinitions>
    <Ribbon>
        <RibbonTab Header="Rechnungsverwaltung">
            <RibbonGroup Header="Kunden">
                <RibbonButton x:Name="btnKundenuebersicht" Label="Übersicht" LargeImageSource="Images/users.png"
                    Click="BtnKundenuebersicht_Click"></RibbonButton>
                <RibbonButton x:Name="btnKundeHinzufuegen" Label="Neuer Kunde" LargeImageSource="Images/user_add.png"
                    Click="BtnKundeHinzufuegen_Click"></RibbonButton>
```

```
            </RibbonGroup>
            <RibbonGroup Header="Rechnungen">
                <RibbonButton x:Name="btnRechnungsuebersicht" Label="Übersicht" LargeImageSource="Images/invoice.png"
                    Click="BtnRechnungsuebersicht_Click"></RibbonButton>
                <RibbonButton x:Name="btnRechnungHinzufuegen" Label="Neue Rechnung"
                    LargeImageSource="Images/invoice_add.png" Click="BtnRechnungHinzufuegen_Click"></RibbonButton>
            </RibbonGroup>
        </RibbonTab>
    </Ribbon>
</Grid>
```

Damit erstellen wir ein Ribbon, das wie in Bild 5 aussieht. Die Ereignismethoden für die Schaltflächen werden automatisch angelegt, wenn Sie im XAML-Code **Click** gefolgt vom Gleichheitszeichen eingeben und dann die Tabulator-Taste betätigen.

Bild 5: Ribbon der Anwendung

Kunden anlegen

Wir beginnen mit der Seite zum Anlegen eines neuen Kunden, da wir ja noch keine Kundendaten in der Datenbank haben – und diese sollten ja vorhanden sein, damit wir Material für die Übersichtsseite der Kunden haben. Wir erstellen also explizit kein neues Fenster, sondern eine Seite (**Page**). Diese fügen wir hinzu, indem wir den Kontextmenü-Befehl des Projekt-Elements im Projektmappen-Explorer betätigen und im Dialog **Neues Element hinzufügen** den Eintrag **Seite (WPF)** auswählen. Die neue Seite soll **Kundendetails** heißen und nicht nur zum Anlegen neuer Kunden, sondern auch zur Anzeige der Details bereits angelegter Kunden dienen. Der XAML-Code sieht in gekürzter Form wie folgt aus:

```
<Page x:Class="Kundendetails" ...>
    <Page.Resources>...</Page.Resources>
    <Grid>
        ...
```

Bei den **TextBox**-Elementen wird der Inhalt des jeweiligen **Kunde**-Elements durch eine Bindung an das Feld **Kunde.ID** erreicht:

```
        <Label Content="ID:" Grid.Column="0" />
        <TextBox x:Name="txtID" Grid.Column="1" HorizontalAlignment="Left"
            Text="{Binding Kunde.ID, Mode=TwoWay}" Width="50" IsEnabled="False" BorderBrush="Transparent" />
```

Die Ausnahme ist hier die das **ComboBox**-Element **cboAnrede**. Es verwendet eine ganze Reihe von Attributen, damit es die Daten der Tabelle **Anreden** anzeigt. **ItemsSource** gibt den Namen der Eigenschaft der Code behind-Datei an, welche die Daten

für das **ComboBox**-Element liefert. **SelectedItem** legt fest, wie der ausgewählte Eintrag selektiert wird. **SelectedValuePath** gibt den Namen des Feldes an, über den das selektierte Element ermittelt wird.

Schließlich gibt das **Binding**-Element im **DataTemplate**-Element noch an, welches Feld der **Anrede**-Objekte im Kombinationsfeld angezeigt wird:

```
<Label Content="ID:" Grid.Column="0" />
<TextBox x:Name="txtID" Grid.Column="1" HorizontalAlignment="Left" Text="{Binding Kunde.ID, Mode=TwoWay}"
    Width="50" IsEnabled="False" BorderBrush="Transparent" />
<Label Content="Anrede:" Grid.Column="0" Grid.Row="1" />
<ComboBox x:Name="cboAnredeID" Grid.Row="1" Grid.Column="1" Width="100"
          ItemsSource="{Binding Anreden}"
          SelectedItem="{Binding Kunde.Anrede, ValidatesOnDataErrors=True}"
          SelectedValuePath="ID" SelectionChanged="Anrede_SelectionChanged">
    <ComboBox.ItemTemplate>
        <DataTemplate>
            <TextBlock>
                <TextBlock.Text>
                    <MultiBinding StringFormat="{}{0}">
                        <Binding Path="Bezeichnung" />
                    </MultiBinding>
                </TextBlock.Text>
            </TextBlock>
        </DataTemplate>
    </ComboBox.ItemTemplate>
</ComboBox>
<Label Content="Vorname:" Grid.Column="0" Grid.Row="2" />
<TextBox x:Name="txtVorname" Grid.Column="1" Grid.Row="2" Width="200"
    Text="{Binding Kunde.Vorname, Mode=TwoWay, ValidatesOnDataErrors=True}" />
<Label Content="Nachname:" Grid.Row="3" />
<TextBox x:Name="txtNachname" Grid.Column="1" Grid.Row="3" Width="200"
    Text="{Binding Kunde.Nachname, Mode=TwoWay, ValidatesOnDataErrors=True}" />
<Label Content="Firma:" Grid.Row="4" />
<TextBox x:Name="txtFirma" Grid.Column="1" Grid.Row="4" Width="200"
    Text="{Binding Kunde.Firma, Mode=TwoWay, ValidatesOnDataErrors=True}" />
<Label Content="Straße:" Grid.Row="5" />
<TextBox x:Name="txtStrasse" Grid.Column="1" Grid.Row="5" Width="200"
    Text="{Binding Kunde.Strasse, Mode=TwoWay, ValidatesOnDataErrors=True}" />
<Label Content="PLZ:" Grid.Row="6" />
<TextBox x:Name="txtPLZ" Grid.Column="1" Grid.Row="6" Width="50"
    Text="{Binding Kunde.PLZ, Mode=TwoWay, ValidatesOnDataErrors=True}" />
```

```
        <Label Content="Ort:" Grid.Row="7" />
        <TextBox x:Name="txtOrt" Grid.Column="1" Grid.Row="7" Width="200"
            Text="{Binding Kunde.Ort, Mode=TwoWay, ValidatesOnDataErrors=True}" />
        <Label Content="Land:" Grid.Row="8" />
        <TextBox x:Name="txtLand" Grid.Column="1" Grid.Row="8" Width="200"
            Text="{Binding Kunde.Land, Mode=TwoWay, ValidatesOnDataErrors=True}" />
        <Label Content="E-Mail:" Grid.Row="9" />
        <TextBox x:Name="txtEMail" Grid.Column="1" Grid.Row="9" Width="200"
            Text="{Binding Kunde.EMail, Mode=TwoWay, ValidatesOnDataErrors=True}" />
        <StackPanel Orientation="Horizontal" Grid.Row="10" Grid.ColumnSpan="4" >
            <Button x:Name="btnSpeichern" Click="btnSpeichern_Click" Content="Speichern">
                <Button.Style>
                    <Style TargetType="{x:Type Button}" BasedOn="{StaticResource {x:Type Button}}">
                        <Style.Triggers>
                            <DataTrigger Binding="{Binding ElementName=cboAnrede,
                                    Path=(Validation.HasError)}" Value="True">
                                <Setter Property="IsEnabled" Value="False"></Setter>
                            </DataTrigger>
                            ...
                        </Style.Triggers>
                    </Style>
                </Button.Style>
            </Button>
            <Button x:Name="btnVerwerfen"
                Click="btnVerwerfen_Click"
                Content="Verwerfen"></Button>
        </StackPanel>
    </Grid>
</Page>
```

Wir betten hier direkt auch die Funktion zum Validieren der Textfelder ein – daher finden Sie im **Binding**-Element auch jeweils die Eigenschaft **ValidationOnDataErrors** mit dem Wert **True**.

Die so erstellte Seite betten wir beim Anklicken des Ribbon-Eintrags **btnKundeAnlegen** über die folgende Methode ein:

```
Private Sub BtnKundeHinzufuegen_Click(sender As Object, e As RoutedEventArgs)
```

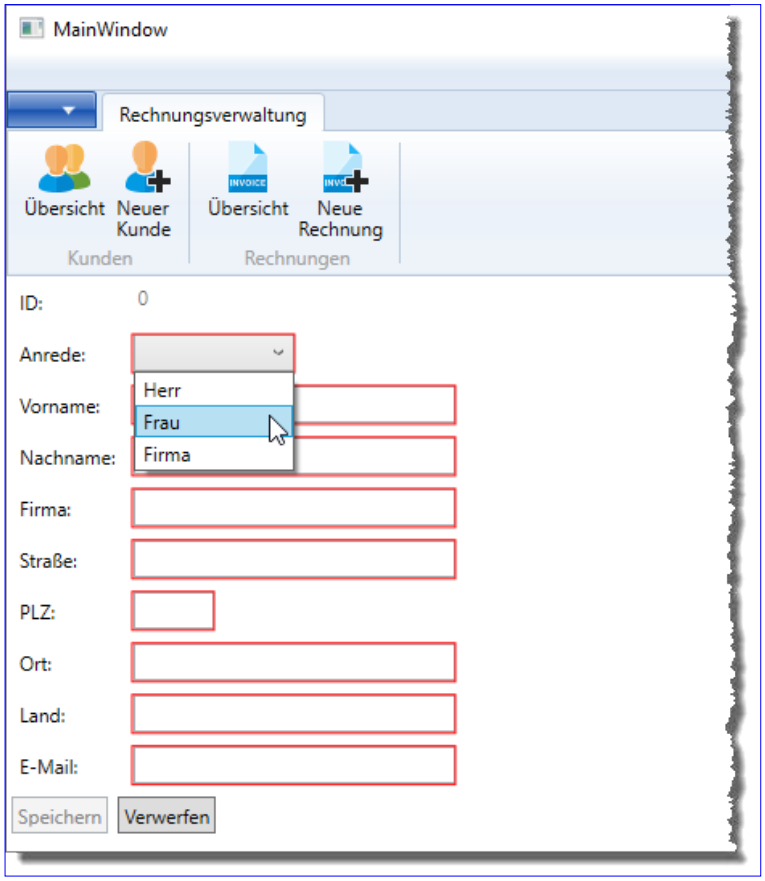


Bild 6: Die Seite **Kundendetails.xaml** im Fenster **Main.xaml**

```
    Dim pge As Kundendetails
    pge = New Kundendetails(Me.WorkZone)
    WorkZone.Content = pge
End Sub
```

Wir erstellen also ein neues Objekt auf Basis des **Page**-Elements **Kundendetails** und übergeben diesem einen Verweis auf das **Frame**-Element **WorkZone**, in dem wir es anzeigen wollen.

Diesen benötigen wir später, um die Seite wieder aus dem **Frame**-Element zu entfernen. Dann weisen wir es dem mit **WorkZone** benannten **Frame**-Element zu, das wir wie folgt im Fenster **Main.xaml** eingefügt haben:

```
<Frame x:Name="WorkZone" Grid.Row="1" NavigationUIVisibility="Hidden" Navigated="WorkZone_Navigated"></Frame>
```

Nach dem Starten des Projekts und einem Klick auf die Ribbon-Schaltfläche **Neuer Kunde** sieht das Fenster wie in Bild 6 aus. Hier wird gerade die Anrede für einen neuen Kunden ausgewählt.

Damit die Daten so wie im Screenshot auf der Seite landen, sind noch einige Zeilen in der Code behind-Datei nötig. Diese sehen wie folgt aus. Die Klasse deklariert zunächst drei Variablen. Die erste namens **_kunde** nimmt den aktuell angezeigten Kunden auf, die zweite namens **_anreden** ist eine Auflistung von Objekten des Typs **Anrede** und die dritte namens **dbContext** referenziert das Context-Objekt:

```
Class Kundendetails
    Private _kunde As Kunde
    Private _anreden As List(Of Anrede)
    Private dbContext As RechnungsverwaltungContext
```

Außerdem benötigen wir noch eine Variable, mit der wir das **Frame**-Objekt referenzieren, in dem das **Page**-Element angezeigt wird:

```
    Private _frame As Frame
```

Die private Variable **_kunde** machen wir über die öffentliche Eigenschaft **Kunde** schreib- und lesbar:

```
    Public Property Kunde As Kunde
        Get
            Return _kunde
        End Get
        Set(value As Kunde)
            _kunde = value
        End Set
    End Property
```

Gleiches erledigen wir für die Auflistung aus der privaten Variablen **_anreden**:

```
Public Property Anreden As List(Of Anrede)
    Get
        Return _anreden
    End Get
    Set(value As List(Of Anrede))
        _anreden = value
    End Set
End Property
```

Es gibt zwei verschiedene Konstruktor-Methoden, die beim Starten abhängig von den übergebenen Parametern aufgerufen werden können. Falls einer der bestehenden Kunden aus der nachfolgend beschriebenen Liste der Kunden aufgerufen wird, übergeben wir dabei den Primärschlüsselwert des jeweiligen Kunden als Parameter. Die Methode **InitializeComponent** erstellt das Fenster auf Basis der **.xaml**-Datei. Die zweite Anweisung schreibt den mit dem Parameter **objFrame** übergebenen Verweis auf das übergeordnete **Frame**-Element in die Variable **_frame**. Die nächste Anweisung initialisiert das Context-Objekt. Dann weisen wir die Code behind-Klasse als Datenquelle für die Bindung der Steuerelemente des **Page**-Elements zu. Und schließlich füllen wir die Eigenschaft **Kunde** mit dem Kunden, der den angegebenen Primärschüsselwert aufweist sowie die Liste **Anreden** mit allen Elementen des **DbSet**-Objekts **Anreden**:

```
Public Sub New(objFrame As Frame, lngKundeID As Long)
    InitializeComponent()
    _frame = objFrame
    dbContext = New RechnungsverwaltungContext
    DataContext = Me
    Anreden = dbContext.Anreden.ToList
    Kunde = dbContext.Kunden.Find(lngKundeID)
End Sub
```

Die zweite Konstruktor-Methode hat keinen zweiten Parameter. Sie wird folglich aufgerufen, wenn wir wie oben auf die Ribbon-Schaltfläche **btnKundeHinzufuegen** klicken. Sie erledigt die gleichen Aufgaben wie die zuvor beschriebene Konstruktor-Methode, holt aber keinen bestehenden Kunden, sondern legt einen neuen an:

```
Public Sub New(objFrame As Frame)
    InitializeComponent()
    _frame = objFrame
    dbContext = New RechnungsverwaltungContext
    DataContext = Me
    Anreden = dbContext.Anreden.ToList
    Kunde = New Kunde
    dbContext.Kunden.Add(Kunde)
```

Rechnungsverwaltung, Teil 2: Rechnungspositionen

Als selbständiger oder freiberuflicher Softwareentwickler braucht man am Ende vor allem eines: eine Anwendung zum Erstellen von Rechnungen. Diese wollen wir im vorliegenden Artikel programmieren – vom Entwurf des Datenmodells über die Erstellung des Entity Data Models und die Benutzeroberfläche bis zum Ausdrucken der Rechnung als PDF oder mit dem Drucker. Im zweiten Teil der Beitragsreihe fügen wir die Verwaltung von Rechnungspositionen hinzu.

Vorüberlegungen

Wenn wir unter Access Rechnungen und Rechnungspositionen verwalten wollten, war das sehr einfach möglich – mit einem Haupt- und einem Unterformular. Einzige Bedingung war, dass der Benutzer den neuen, leeren Datensatz im Hauptformular bereits bearbeitet hatte, damit dieser einen Autowert für sein Primärschlüsselfeld erhalten hat. Damit konnte man dann im Unterformular, dessen Datensätze über das Fremdschlüsselfeld mit dem entsprechenden Datensatz im Hauptformular verknüpft waren, direkt die Bestellpositionen eingeben.

Unter .NET mit WPF und dem Entity Framework ist das ein wenig aufwendiger. In diesem Beispiel wollen wir es so gestalten, dass wir ein Listenfeld zur Anzeige der vorhandenen Rechnungspositionen verwenden, die jeweils zusammen mit den Rechnungsdetails angezeigt werden. Das sieht in der Entwurfsansicht zunächst wie in Bild 1 aus. Von hier aus wollen wir ein weiteres Fenster zur Eingabe von Rechnungspositionen öffnen. Dieses enthält dann jeweils die Daten einer Rechnungsposition. Da diese an eine Rechnung gebunden sein muss, benötigen wir vor dem Speichern einer Rechnungsposition ein Rechnungsobjekt, mit dem diese Rechnung verknüpft ist oder zumindest den Primärschlüsselwert dieser Rechnung. Anderenfalls können wir die Rechnungsposition nicht speichern, da das Fremdschlüsselfeld zum Referenzieren der Rechnung nicht gefüllt wäre.

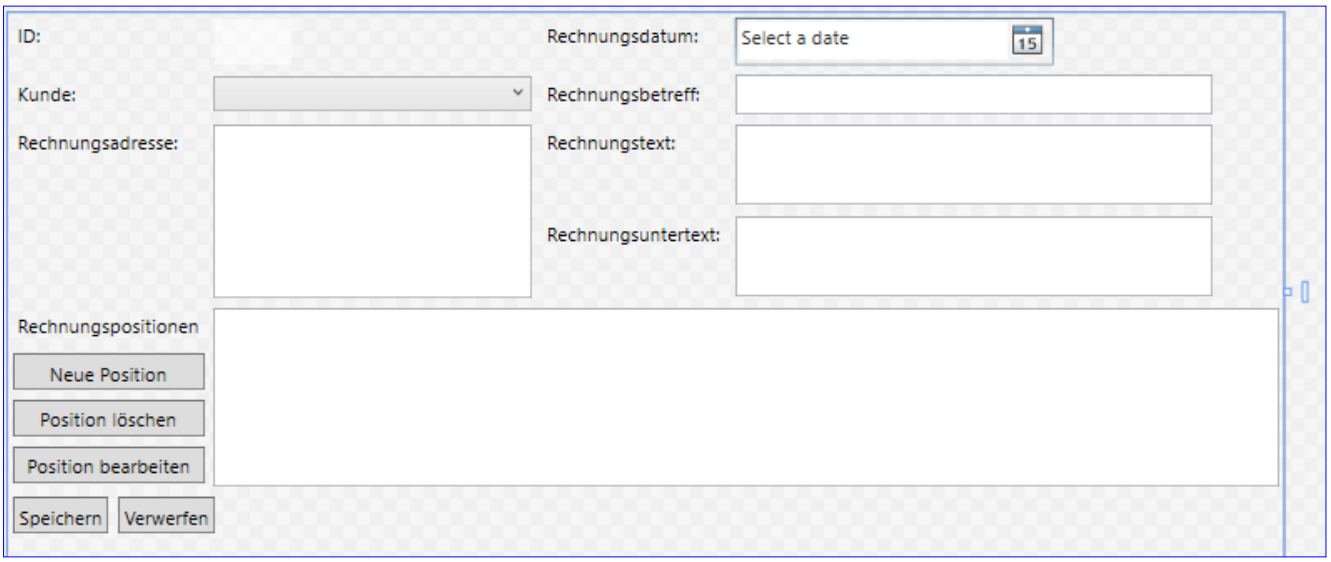


Bild 1: Erweiterung der Seite zur Anzeige der Rechnungsdetails

Bei Access reichte es aus ein Feld eines Datensatzes mit einem Wert zu füllen, um die Vergabe eines Autowerts für das Primärschlüsselfeld auszulösen. Bei dem darauf folgenden Wechsel in das Unterformular, wurde dieser Datensatz automatisch gespeichert und der Primärschlüssel stand ab sofort als Fremdschlüssel für die Anlage der untergeordneten Datensätze zur Verfügung. Unter WPF existiert dieser Automatismus nicht, weshalb wir das gewünschte Verhalten selber entwickeln müssen.

Wir haben unter anderem die folgenden Möglichkeiten:

- Hinzufügen einer Schaltfläche, mit welcher der Benutzer den Datensatz manuell speichern kann und damit die Steuerelemente zum Bearbeiten der Rechnungspositionen freigibt,

- automatisches Speichern der Rechnungsdaten, wenn der Benutzer die Schaltfläche zum Anlegen einer Rechnungsposition betätigt.

Beides ist ohnehin an die Validierung gebunden: Bevor der Rechnungsdatensatz nicht vollständig gefüllt wurde, kann dieser nicht gespeichert werden. Das bedeutet, dass wir auch die Freigabe der Schaltfläche zum Anlegen einer neuen Position an das vollständige Ausfüllen der Rechnungsdaten binden können.

Wir wollen den zweiten Weg gehen und das Erstellen der ersten Rechnungsposition erst dann freigeben, wenn der Benutzer alle Rechnungsdaten eingegeben hat – und den Rechnungsdatensatz dann automatisch speichern. Mehr dazu weiter unten – erst folgen noch ein paar Vorbereitungen.

Berechnete Felder vorbereiten

Wir wollen für die Rechnungspositionen sowohl den Bruttobetrag des Einzelpreises sowie den gesamten Bruttobetrag ausgeben. Dazu benötigen wir zwei berechnete Felder namens **Brutto** und **BruttoGesamt**. Diese beiden Felder stecken nicht in den zugrunde liegenden Tabellen, sondern sie sollen ausschließlich in der Anwendung berechnet werden.

Dazu könnten wir einfach der Klasse **Rechnungsposition** zwei Eigenschaften hinzufügen, welche die Werte auf Basis der Felder **Netto**, **Mehrwertsteuersatz** und **Menge** ermittelt. Allerdings kann es immer mal passieren, dass wir das Datenmodell ändern und die Klassen auf das neue Datenmodell anpassen wollen. Wenn wir das nicht von Hand erledigen, sondern beispielsweise auf Basis des Access-Datenmodells, wie wir es im ersten Teil der Artikelreihe verwendet haben, werden manuelle Änderungen an den Entitätsklassen jedes Mal überschrieben. Also nutzen wir die Möglichkeit, partielle Klassen anzulegen und erstellen eine neue Klasse namens **Rechnungsposition_Calc**. Diese erhält den folgenden Code:

```
Partial Public Class Rechnungsposition
    Implements INotifyPropertyChanged

    Public ReadOnly Property Brutto As System.Decimal
        Get
            Return _Netto * (1 + _Mehrwertsteuersatz / 100)
        End Get
    End Property
```

```
    Public ReadOnly Property BruttoGesamt As System.Decimal
        Get
            Return _Netto * (1 + _Mehrwertsteuersatz / 100) * Menge
        End Get
    End Property
End Class
```

Die **Property**-Eigenschaften unterscheiden sich in verschiedenen Punkten von denen, die wir in der Klasse **Rechnungsposition.vb** vorfinden:

- Sie liefern nicht die Werte von Feldern der zugrunde liegenden Tabellen zurück, sondern berechnete Felder auf Basis mehrerer Tabellenfelder.
- Sie weisen nur eine **Get**-Property, aber keine **Set**-Property auf. Das ist logisch, denn sie sollen ja nicht vom Benutzer eingestellt werden können, sondern werden nur berechnet.
- Da sie nur Werte zurückliefern, müssen wir das Schlüsselwort **ReadOnly** verwenden.

Die Eigenschaft **Brutto** ermitteln wir als das Produkt des Nettopreises mit dem Wert **1** plus dem Prozentsatz aus **Mehrwertsteuersatz** geteilt durch **100**. Für die Eigenschaft **BruttoGesamt** multiplizieren wir dies noch mit dem Wert des Feldes **Menge**.

Aktualisieren der Felder, wenn die zugrunde liegenden Felder aktualisiert werden

Die Implementierung der Schnittstelle **INotifyPropertyChanged** sorgt in den Entitätsklassen dafür, dass Steuerelemente, die an Felder dieser Klassen gebunden sind, bei Änderungen der Feldinhalte automatisch aktualisiert werden. Das ist bei den berechneten Eigenschaften nicht der Fall, da diese ja keine **Set**-Property und damit auch keine Möglichkeit für den Einbau des Aufrufs von **OnPropertyChanged** bieten.

Hier müssen wir nun doch noch in den Code der automatisch generierten Entitätsklassen eingreifen. Wir fügen den Eigenschaften, auf denen die berechneten Eigenschaften basieren, Aufrufe der Methode **OnPropertyChanged** hinzu, welche nicht nur die Eigenschaft selbst, sondern auch die berechneten Eigenschaften aktualisieren, die auf dieser Eigenschaft aufbauen:

```
Public Property Netto As System.Decimal
    Get
        Return _Netto
    End Get
    Set
        _Netto = Value
        OnPropertyChanged("Netto")
        OnPropertyChanged("Brutto")
        OnPropertyChanged("BruttoGesamt")
```

```
    End Set
End Property
```

Auf die gleiche Weise erweitern wir auch die **Property Set**-Eigenschaften **Menge** und **Mehrwertsteuersatz**. Das Ergebnis: Textfelder, welche die berechneten Felder **Brutto** und **BruttoGesamt** anzeigen, werden nun bei Änderung der Werte der Felder **Netto**, **Anzahl** und **Mehrwertsteuersatz** ebenfalls aktualisiert. Wenn Sie tatsächlich einmal das Datenmodell ändern und die Entitätsklassen neu generieren lassen, müssen Sie diese Aufrufe von **OnPropertyChanged** manuell wiederherstellen.

Steuerelemente zum Verwalten der Rechnungspositionen

Zum Verwalten der Rechnungspositionen wollen wir folgende Steuerelemente hinzufügen:

- Listenfeld zur Anzeige der vorhandenen Rechnungspositionen
- Schaltfläche zum Hinzufügen einer Rechnungsposition
- Schaltfläche zum Löschen einer Rechnungsposition
- Schaltfläche zum Bearbeiten der aktuell markierten Rechnungsposition

Außerdem soll das Listenfeld per Doppelklick auf einen Eintrag erlauben, die Rechnungsposition zu bearbeiten.

Zum Anlegen und Bearbeiten einer Rechnungsposition soll ein Popup-Fenster mit den Details der Rechnungsposition geöffnet werden. Dieses nennen wir **RechnungspositionDetail.xaml**, das im Entwurf wie in Bild 2 aussehen soll.

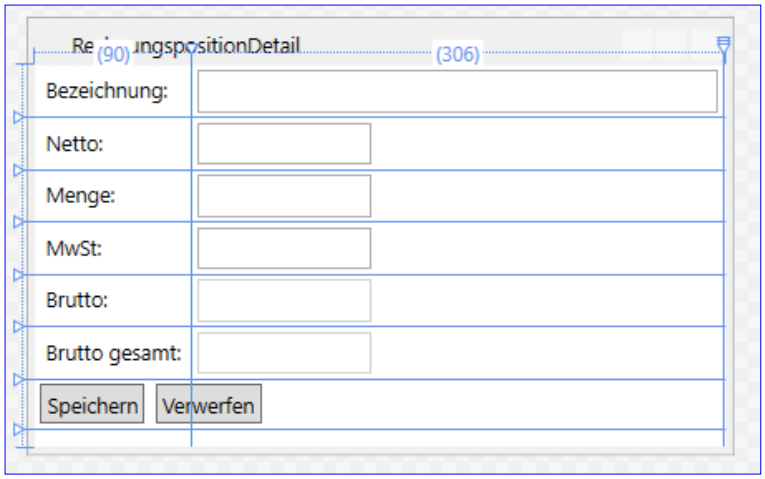


Bild 2: Entwurf des Fensters zur Eingabe einer Rechnungsposition

RechnungspositionDetail.xaml wollen wir nicht als Seite, sondern als Fenster anlegen. Auf diese Weise ist das Hauptfenster mit der Rechnung, für die Rechnungspositionen angelegt werden sollen, immer noch sichtbar und der Benutzer kann prüfen, welche Rechnungspositionen er bereits angelegt hat.

Damit das Fenster in der Mitte angezeigt wird und nicht irgendwo, stellen wir in **RechnungspositionenDetail.xaml** für das **Window**-Element das Attribut **WindowStartupLocation** auf **CenterScreen** ein:

```
<Window x:Class="RechnungspositionDetail"...
        Title="RechnungspositionDetail" Height="250" Width="405"
        WindowStartupLocation="CenterScreen">
```

Danach definieren wir die Zeilen und Spalten des Grids:

```
    <Grid>
        <Grid.ColumnDefinitions>
            <ColumnDefinition Width="Auto"></ColumnDefinition>
            ... weitere Spalten
            <ColumnDefinition Width="*"></ColumnDefinition>
        </Grid.ColumnDefinitions>
        <Grid.RowDefinitions>
            <RowDefinition Height="Auto"></RowDefinition>
            ... weitere Zeilen
            <RowDefinition Height="*"></RowDefinition>
        </Grid.RowDefinitions>
```

Die Steuerelemente, hier ausschließlich Textfelder, binden wir an Elemente des Objekts **Rechnungsposition** und stellen ein entsprechendes Label-Steuerelement voran:

```
        <Label>Bezeichnung:</Label>
        <TextBox x:Name="txtBezeichnung" Width="300" Grid.Column="1" Text="{Binding Rechnungsposition.Bezeichnung,
Mode=TwoWay, ValidatesOnDataErrors=True}"></TextBox>
```

Für das Feld **Netto** legen wir das Format mit **StringFormat=C** fest und sorgen so für die Anzeige des Betrags mit Euro-Zeichen:

```
        <Label Grid.Row="1">Netto:</Label>
        <TextBox x:Name="txtNetto" Width="100" Grid.Row="1" Grid.Column="1" Text="{Binding Rechnungsposition.Netto,
            StringFormat=C, Mode=TwoWay, ValidatesOnDataErrors=True}"></TextBox>
```

Abhängig von den Ländereinstellungen taucht hier dennoch das Dollar-Zeichen als Währungssymbol auf. Um dies zu ändern, fügen Sie der Anwendung für die Datei **Application.xaml.vb** die folgende Methode hinzu, die beim Starten der Anwendung aufgerufen wird. Außerdem benötigen wir dazu die folgenden beiden **Namespace**-Verweise:

```
Imports System.Globalization
Imports System.Windows.Markup

Class Application
    Protected Overrides Sub OnStartup(ByVal e As StartupEventArgs)
        FrameworkElement.LanguageProperty.OverrideMetadata(GetType(FrameworkElement),
            New FrameworkPropertyMetadata(XmlLanguage.GetLanguage(CultureInfo.CurrentCulture.IetfLanguageTag)))
        MyBase.OnStartup(e)
    End Sub
End Class
```

Das Feld **Menge** benötigt keine weitere Formatierung:

```
<Label Grid.Row="2">Menge:</Label>
<TextBox x:Name="txtMenge" Width="100" Grid.Row="2" Grid.Column="1"
    Text="{Binding Rechnungsposition.Menge, Mode=TwoWay, ValidatesOnDataErrors=True}"></TextBox>
```

Für das Feld **Mehrwertsteuersatz** definieren wir das Format mit **StringFormat={}{0} %**. Die vorgegebenen Formate multiplizieren den Wert jeweils mit 100. Sie müssten also die Prozentzahl für 1% mit 0,01 angeben. Das wäre korrekt und ließe sich ebenfalls abbilden, aber hier wählen wir die einfachere Variante – wir müssen dies nur beim Berechnen des Brutto-Preises berücksichtigen, was wir oben in den berechneten Feldern bereits erledigt haben:

```
<Label Grid.Row="3">MwSt:</Label>
<TextBox x:Name="txtMehrwertsteuersatz" Width="100" Grid.Row="3" Grid.Column="1"
    Text="{Binding Rechnungsposition.Mehrwertsteuersatz, StringFormat={}{0} %, Mode=TwoWay,
ValidatesOnDataErrors=True}"></TextBox>
```

Die beiden Felder **Brutto** und **BruttoGesamt** sind die berechneten Felder. Damit der Benutzer diese Werte nicht direkt ändern kann, deaktivieren wir die entsprechenden Textfelder durch Einstellen des Attributs **IsEnabled** auf den Wert **False**. Davon abgesehen greifen wir wie üblich auf die Felder **Brutto** und **BruttoGesamt** zu:

```
<Label Grid.Row="4">Brutto:</Label>
<TextBox x:Name="txtBrutto" Grid.Row="4" Width="100" Grid.Column="1" IsEnabled="False"
    Text="{Binding Rechnungsposition.Brutto, Mode=OneWay, StringFormat=C}"></TextBox>
<Label Grid.Row="5">Brutto gesamt:</Label>
<TextBox x:Name="txtBruttoGesamt" Grid.Row="5" Width="100" Grid.Column="1" IsEnabled="False"
    Text="{Binding Rechnungsposition.BruttoGesamt, Mode=OneWay, StringFormat=C}"></TextBox>
```

Die beiden Schaltflächen zum Speichern und Verwerfen fassen wir in ein **StackPanel**-Element ein. Die Schaltfläche **btnSpeichern** soll nur aktiviert werden, wenn die Validierungen für die einzugebenden gebundenen Felder erfolgreich sind. Daher fügen wir wieder einen DataTrigger hinzu, der die Validierung der Felder **txtBezeichnung**, **txtNetto**, **txtMenge** und **txtMehrwertsteuer** durchführt und im Falle auch nur eines negativen Ergebnisses die Schaltfläche **btnSpeichern** deaktiviert:

```
<StackPanel Orientation="Horizontal" Grid.Row="6" Grid.ColumnSpan="4">
    <Button x:Name="btnSpeichern" Click="btnSpeichern_Click" Content="Speichern">
        <Button.Style>
            <Style TargetType="{x:Type Button}" BasedOn="{StaticResource {x:Type Button}}">
                <Style.Triggers>
                    <DataTrigger Binding="{Binding ElementName=txtBezeichnung,
                            Path=(Validation.HasError)}" Value="True">
                        <Setter Property="IsEnabled" Value="False"></Setter>
                    </DataTrigger>
```

```
                        <DataTrigger Binding="{Binding ElementName=txtNetto,
                                Path=(Validation.HasError)}" Value="True">
                            <Setter Property="IsEnabled" Value="False"></Setter>
                        </DataTrigger>
                        <DataTrigger Binding="{Binding ElementName=txtMenge,
                                Path=(Validation.HasError)}" Value="True">
                            <Setter Property="IsEnabled" Value="False"></Setter>
                        </DataTrigger>
                        <DataTrigger Binding="{Binding ElementName=txtMehrwertsteuer,
                                Path=(Validation.HasError)}" Value="True">
                            <Setter Property="IsEnabled" Value="False"></Setter>
                        </DataTrigger>
                    </Style.Triggers>
                </Style>
            </Button.Style>
        </Button>
        <Button x:Name="btnVerwerfen" Click="btnVerwerfen_Click" Content="Verwerfen"></Button>
    </StackPanel>
  </Grid>
</Window>
```

Code des Fensters RechnungspositionDetail.xaml

Der größte Teil der Funktionen des Fensters wird über die XAML-Definitionen abgedeckt. So bleibt noch das Füllen der anzuzeigenden oder neuen Rechnungsposition. Die dafür notwendigen Elemente werden oben in der Klasse deklariert:

```
Imports System.ComponentModel

Public Class RechnungspositionDetail
    Private dbContext As RechnungsverwaltungContext
    Private _rechnungsposition As Rechnungsposition
```

Zum Bereitstellen der Rechnungsposition nach außen verwenden wir die folgende Property:

```
    Public Property Rechnungsposition As Rechnungsposition
        Get
            Return _rechnungsposition
        End Get
        Set(value As Rechnungsposition)
            _rechnungsposition = value
        End Set
    End Property
```

Es gibt zwei Konstruktor-Methoden. Die erste wird aufgerufen, wenn das Fenster eine bereits vorhandene Rechnungsposition anzeigen soll. Sie erwartet einen Verweis auf den aktuellen Datenbank-Context sowie das anzuzeigende Rechnungsposition-Objekt als Parameter. Warum erstellen wir keinen neuen Verweis für **dbContext**, wie wir es sonst immer machen? Weil wir hier eine Rechnungsposition übergeben, die aus dem Datenbank-Context der aufrufenden Instanz stammt – und diese können wir nicht mit den Methoden eines neu erstellten Datenbank-Contextes behandeln. Also weisen wir **dbContext** den per Parameter gelieferten Datenbank-Context zu und **Rechnungsposition** die mit **AktuelleRechnungsposition** gelieferte Rechnungsposition:

```
Public Sub New(AktuellerContext As RechnungsverwaltungContext, AktuelleRechnungsposition As Rechnungsposition)
    InitializeComponent()
    dbContext = AktuellerContext
    Rechnungsposition = AktuelleRechnungsposition
    DataContext = Me
End Sub
```

Die zweite Konstruktor-Methode erwartet auch einen Verweis auf den Datenbank-Context, aber als zweiten Parameter die Rechnung, der die Rechnungsposition hinzugefügt werden soll. Wir erstellen eine neue Rechnungsposition und weisen dieser gleich die mit **AktuelleRechnung** übergebene Rechnung für die Eigenschaft **Rechnung** zu. Dann fügen wir sie zur Auflistung der Rechnungspositionen hinzu:

```
Public Sub New(AktuellerContext As RechnungsverwaltungContext, AktuelleRechnung As Rechnung)
    InitializeComponent()
    dbContext = AktuellerContext
    DataContext = Me
    Rechnungsposition = New Rechnungsposition
    Rechnungsposition.Rechnung = AktuelleRechnung
    dbContext.Rechnungspositionen.Add(Rechnungsposition)
End Sub
```

Beim Klick auf die Schaltfläche **btnSpeichern** soll einfach nur die **SaveChanges**-Methode des Datenbank-Contextes aufgerufen und das Fenster geschlossen werden:

```
Private Sub btnSpeichern_Click(sender As Object, e As RoutedEventArgs)
    dbContext.SaveChanges()
    Close()
End Sub
```

Ein Klick auf die Schaltfläche **btnVerwerfen** schließt einfach das Fenster:

```
Private Sub btnVerwerfen_Click(sender As Object, e As RoutedEventArgs)
    Close()
End Sub
```

Rechnungsbericht mit XAML

Wenn Sie einen Rechnungsbericht erstellen wollen, ergeben sich unter .NET zahlreiche Möglichkeiten. Leider ziehen die meisten davon Kosten nach sich in Form von Lizenzgebühren für professionelle Softwareprodukte. Wir wollen in diesem Artikel zeigen, wie Sie einfache Berichte wie etwa für eine Rechnung mit den Bordmitteln von Visual Studio erstellen können. Dazu programmieren wir ein XAML-Fenster, das genau die Größe einer DIN A4-Seite hat und fügen dieser die Steuerelemente zum Anzeigen der für eine Rechnung wichtigen Informationen hinzu. Schließlich programmieren wir auch noch eine Drucken-Funktion, um den Rechnungsbericht zu Papier zu bringen.

Ziel des Artikels

Wir wollen vorab definieren, wie der gewünsche Rechnungsbericht aussehen soll. Das Ergebnis sehen Sie in Bild 1. Der Bericht soll die folgenden Merkmale enthalten:

- Logo rechts oben
- Adressblock, der sowohl die Absenderadresse in einer Zeile als auch die Empfängeradresse enthält
- Informationen zum Rechnungsschreiben, in der die Absenderadresse nochmals wiederholt wird und in der sich weitere Informationen finden –zum Beispiel das Rechnungsdatum. Das kann man beliebig erweitern – um Rechnungsnummern, Auftragsnummern, Ansprechpartner, Geschäftszeichen et cetera.
- RechnungsbetreffRechnungstext, der einleitet und angibt, worauf sich die nachfolgende Aufstellung bezieht
- Aufstellung der Rechnungspositionen
- Summen der Nettopreise, der Mehrwertsteuer und der Bruttopreise
- Rechnungsuntertext, der eine Floskel sowie eine Grußformel enthält

Einige der Elemente des Berichts sind statisch und werden einmalig festgelegt. Daher ist dieses Dokument nur für die Rechnung einer Person oder eines Unternehmens gedacht. Es ist aber auch möglich, mehrere Vorlagen zu erstellen, von denen der Benutzer dann eine auswählt – darum kümmern wir uns aber nicht in diesem Artikel.

Die dynamischen Elemente sind die Daten, die wir aus der in den Artikeln **Rechnungsverwaltung, Teil 1: Grundlagen** (**www.datenbankentwickler.net/221**) und **Rechnungsverwaltung, Teil 2: Rechnungspositionen** (**www.datenbankentwickler.net/222**) vorgestellt haben. In die dort vorgestellte Anwendung binden wir die Erstellung des Rechnungsberichts auch ein. Dazu fügen wir dem Fenster **Rechnungsdetails.xaml** drei Schaltflächen wie in Bild 2 hinzu. Die Schaltflächen haben die folgenden Funktionen:

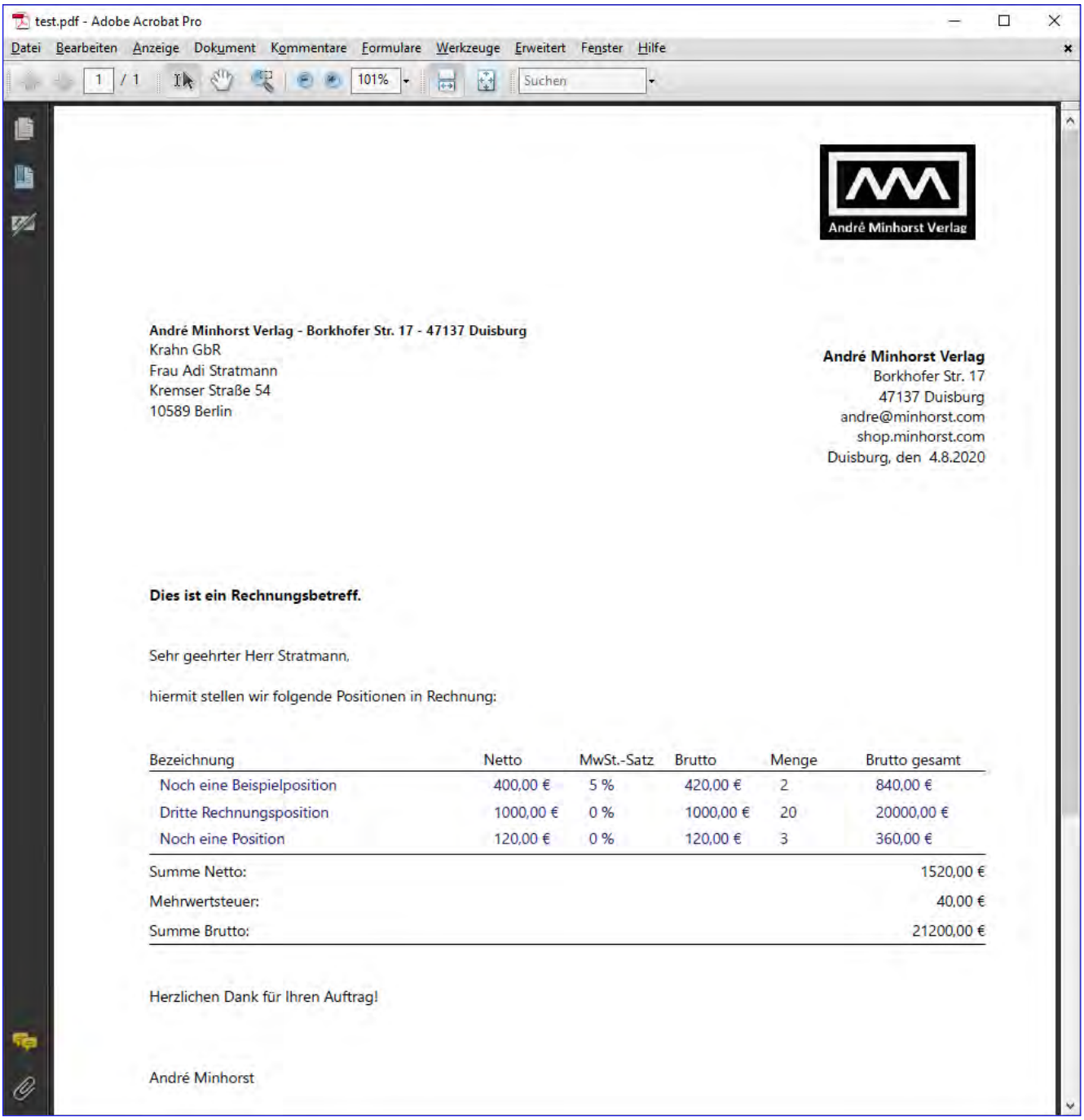


Bild 1: Der geplante Rechnungsbericht

- **Rechnung XAML anzeigen**: Öffnet die Rechnung als Fenster.
- **Rechnung XAML drucken**: Druckt die Rechnung auf dem Standarddrucker aus.
- **Rechnung XAML drucken mit Dialog**: Öffnet den **Drucken**-Dialog.

Bevor wir uns die Funktion dieser Schaltflächen ansehen, bauen wir die Seite zur Anzeige der Rechnung zusammen.

XAML-Seite als Rechnung

Wir haben eigentlich vorgehabt, den Rechnungsbericht mit dem **FlowDocument**-Element zu programmieren. Das hat allerdings nicht wie gewünscht funktioniert, da dieses Element nicht die Möglichkeit geboten hat, Elemente absolut zu positionieren. Dann haben wir ein wenig mit einem herkömmlichen XAML-Fenster experimentiert und gesehen, dass dieses alle Möglichkeiten bietet, die Elemente einer Rechnung abzubilden – inklusive Datenbindung. Also haben wir diese Variante als Basis für diesen Artikel gewählt.

Mit dem XAML-Fenster können wir, wenn wir es einmal auf eine Größe von 21,7cm Breite und 29cm Höhe gebracht haben, ganz einfach die gewünschten Elemente positionieren. Um das zu erreichen, haben wir einfach die beiden Eigenschaften **Height** und **Width** auf diese Werte eingestellt:

```
<Window x:Class="Rechnungsbericht_XAML"
        ...
        Title="Rechnungsbericht_XAML" Height="29.7cm" Width="21cm">
```

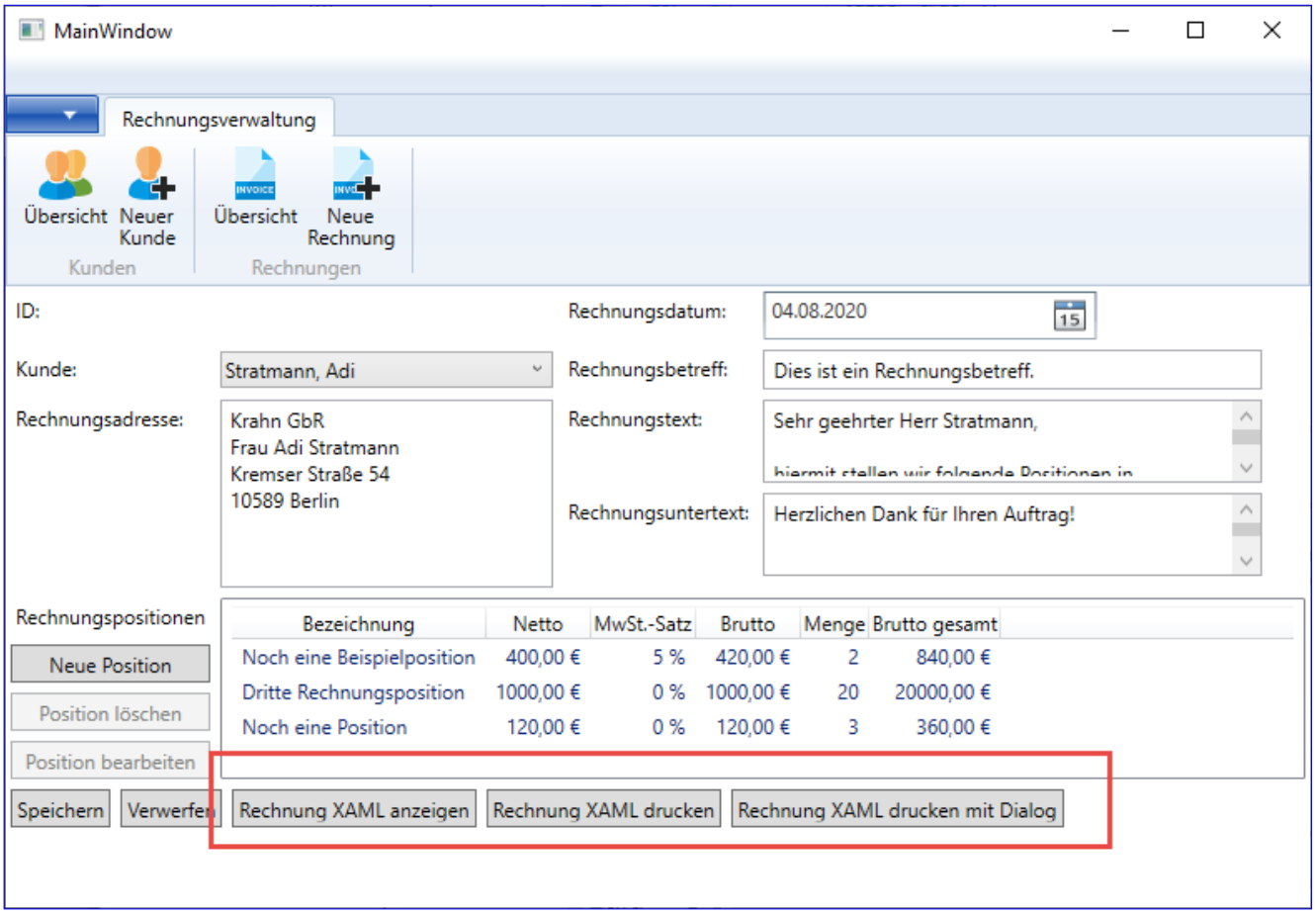


Bild 2: Schaltflächen zum Anzeigen der Rechnung, zum Schnelldrucken und zum Anzeigen des Drucken-Dialogs

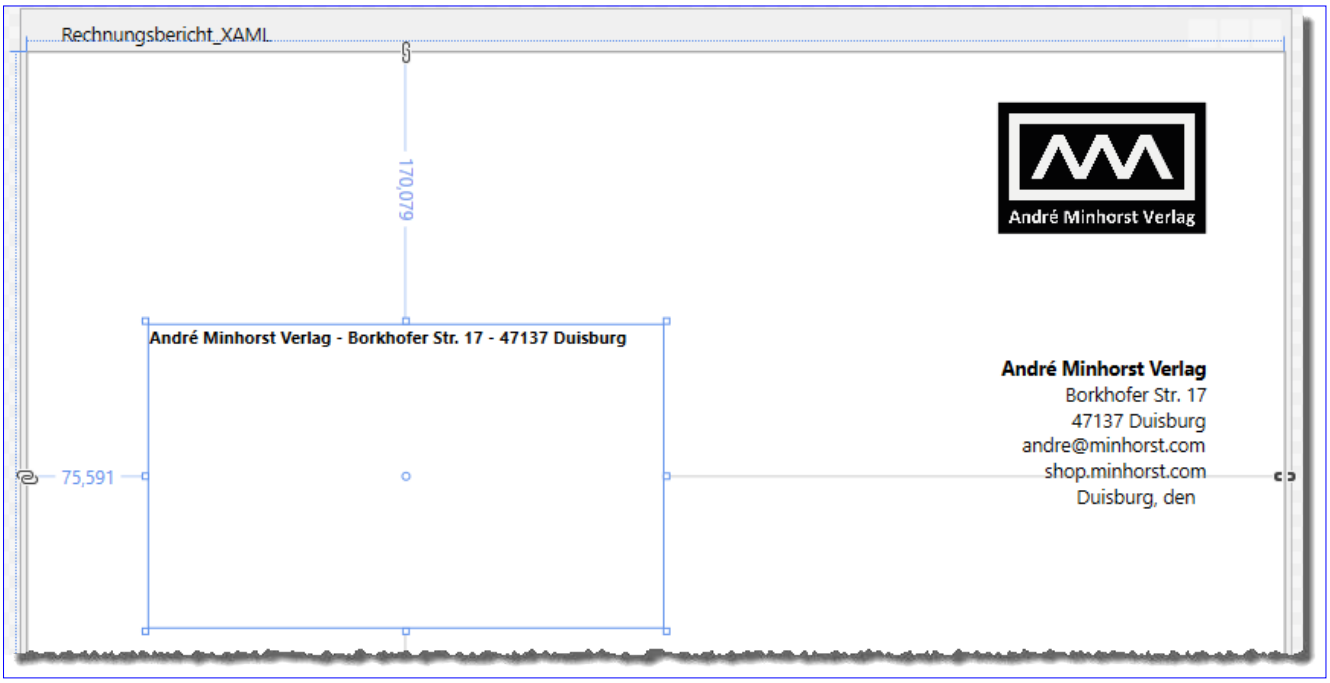


Bild 3: Hinzufügen des Adressblocks

Ein wenig Flow benötigen wir allerdings auch ohne FlowDocument in unserem XAML-Fenster: Wir verwenden ein **ListView**-Steuerelement, um die Rechnungspositionen darzustellen. Rechnungen weisen aber unterschiedlich viele Rechnungspositionen auf, sodass wir an dieser Stelle ein wenig tricksen mussten: Wir haben also die Elemente, die sich unterhalb des **ListView**-Steuerelements befinden und die gegebenenfalls mal mehr, mal weniger weit unten positioniert werden müssen, als Textblöcke in einem **StackPanel**-Element mit vertikaler Orientierung angeordnet, dessen Elemente nach unten geschoben werden, wenn ein darüber befindliches Element vergrößert wird. Doch eins nach dem anderen ...

Hinzufügen des Adressblocks

Um den Adressblock einzufügen, legen wir als erstes Element unterhalb des obligatorischen **Grid**-Elements ein **StackPanel**-Element mit vertikaler Orientierung an (siehe Bild 3).

Dessen Position legen wir über die Eigenschaften **HorizontalAlignment**, **VerticalAlignment**, **Margin**, **Width** und **Height** an. Die Ausrichtung soll links oben sein. Den Abstand vom linken und oberen Seitenrand stellen wir mit den Werten **2cm** und **4.5cm** für die Eigenschaft **Margin** ein. Durch die Eingabe der Zentimeterangaben können wir einfach die Vorgaben etwa der DIN-Norm 5008 umsetzen. Die Breite und die Höhe wollen wir nicht als letzte Werte für die Eigenschaft **Margin** eingeben, da wir diese dann berechnen müssten – wir verwenden stattdessen die Eigenschaften **Height** und **Width**.

Auf diese Weise erhalten wir schon einmal einen Rahmen für unser Adressfeld:

```
<Grid x:Name="GridRechnung">
    <StackPanel Orientation="Vertical" HorizontalAlignment="left" VerticalAlignment="top" Margin="2cm 4.5cm 0cm 0cm"
            Height="5cm" Width="8.5cm">
```

Danach folgt ein **TextBlock**-Element, mit dem wir die Absenderadresse ausgeben wollen. Diese Information ist aktuell hartkodiert, aber wir können diese natürlich auch über eine entsprechende Tabelle verfügbar machen:

```
<TextBlock FontWeight="Bold" FontSize="8pt">André Minhorst Verlag - Borkhofer Str. 17 - 47137 Duisburg</TextBlock>
```

Unter dieser Zeile wollen wir eine optische Trennung in Form einer Linie unterbringen. Mit den Attributen **X1** und **X2** geben wir die X-Koordinaten der Linie gemessen am übergeordneten Element, hier dem **StackPanel**, an. **Stroke** und **StrokeThickness** legen die Farbe und die Dicke fest:

```
<Line X1="0cm" X2="8.5cm" Stroke="Black" StrokeThickness="1"></Line>
```

Schließlich folgt noch ein gebundenes **TextBlock**-Element. Dieses liest per Binding die Eigenschaft **Rechnungsadresse** der Rechnung ein, die über das Code behind-Modul des Fensters bereitgestellt wird – dazu später mehr:

```
    <TextBlock Text="{Binding Rechnung.Rechnungsadresse}"></TextBlock>
</StackPanel>
```

Informationsblock hinzufügen

Den Informationsblock gestalten wir weitgehend frei. Dabei verwenden wir einige fixe Elemente und ein dynamisches – in diesem Fall das Datum, das wir an den Ausdruck **Duisburg, den** anhängen. Die Elemente fügen wir wieder in ein **StackPanel**-Element ein, das wir entsprechend positionieren und dessen Größe wir auf 7 x 5cm anpassen. Darin fügen wir einige

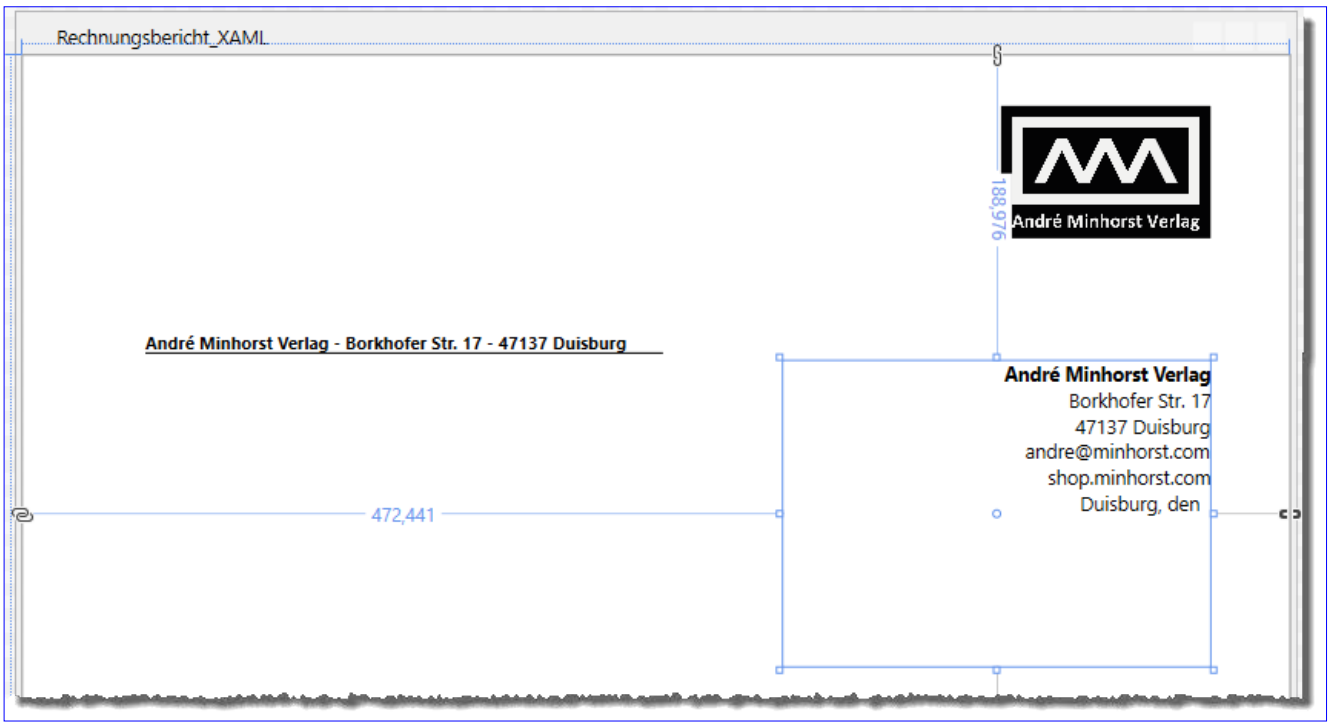


Bild 4: Hinzufügen des Informationsblocks

TextBlock-Elemente ein, von denen jedes einer Zeile des Informationsblocks entspricht. Wir könnten auch ein mehrzeiliges **TextBlock**-Element verwenden, aber dann hätten wir nicht die Möglichkeit, individuelle Formatierungen je Zeile einzufügen. In diesem Fall soll der Firmenname beispielsweise fett dargestellt werden (siehe Bild 4).

```
<StackPanel Orientation="Vertical" HorizontalAlignment="Left" VerticalAlignment="Top" Width="7cm" Height="5cm"
Margin="12.5cm,5cm,0,0">
    <TextBlock FontWeight="Bold" HorizontalAlignment="Right">André Minhorst Verlag</TextBlock>
    <TextBlock HorizontalAlignment="Right">Borkhofer Str. 17</TextBlock>
    <TextBlock HorizontalAlignment="Right">47137 Duisburg</TextBlock>
    <TextBlock HorizontalAlignment="Right">andre@minhorst.com</TextBlock>
    <TextBlock HorizontalAlignment="Right">shop.minhorst.com</TextBlock>
```

Für Teile, die wir aus statischen und dynamischen Texten kombinieren wollen, verwenden wir innerhalb des **TextBlock**-Elements das **Run**-Element. Damit geben wir erst den statischen Text aus und dann den per Binding ermittelten dynamischen Text mit dem angegebenen Format:

```
    <TextBlock HorizontalAlignment="Right">
        <Run Text="Duisburg, den "></Run>
        <Run Text="{Binding Rechnung.Rechnungsdatum, StringFormat='d.M.yyyy'}"></Run>
    </TextBlock>
</StackPanel>
```

Block mit den übrigen Elementen

Die übrigen Elemente starten von einer bestimmten Position beziehungsweise in einem bestimmten Abstand vom oberen Seitenrand aus und bauen dann abhängig von der jeweiligen Höhe aufeinander auf (siehe Bild 5).

Deshalb fügen wir alle weiteren Elemente in ein weiteres **StackPanel**-Element ein, das wieder die vertikale Ausrichtung verwendet. Das erste im **StackPanel**-Element enthaltene **TextBlock**-Element bindet dabei an das Feld **Rechnung.Betreff** der Code behind-Klasse:

```
<StackPanel Orientation="Vertical" HorizontalAlignment="Left" VerticalAlignment="Top" Margin="2cm 10cm 0cm 0cm">
    <TextBlock FontWeight="Bold" Text="{Binding Rechnung.Betreff}"></TextBlock>
```

Danach folgt ein weiteres **TextBlock**-Element, für das wir über die **Padding**-Eigenschaft einen Abstand von einem Zentimeter zum darüber liegenden Element festgelegt haben. Dieses Element ist an das Element **Rechnung.Rechnungstext** der Code behind-Klasse gebunden und zeigt somit den Rechnungstext an:

```
    <TextBlock Text="{Binding Rechnung.Rechnungstext}" Padding="0 1cm 0 0"></TextBlock>
```

Im gleichen Abstand zum vorherigen Element haben wir das **ListView**-Steuerelement namens **lvwRechnungspositionen** festgelegt. Dieses ist an das Feld **Rechnung.Rechnungspositionen** der Code behind-Klasse gebunden und greift somit auf die

Rechnungsverwaltung auf SQLite umstellen

In der Artikelreihe zum Thema Rechnungsverwaltung haben wir ein Rechnungsverwaltungsprogramm mit SQL Server-Datenbank programmiert. SQL Server ist ein sehr zuverlässiges und gutes System für viele Anwendungen. Wenn wir aber eine kleine Rechnungsverwaltung nutzen wollen, die auch bei Freiberuflern oder Selbständigen ohne SQL Server auf dem Rechner laufen soll, benötigen wir eine Alternative. Hier kommt SQLite ins Spiel: eine sehr leichte Datenbanklösung, die noch nicht einmal eine Installation erfordert. Dieser Artikel zeigt, wir wir eine SQLite-Variante von unserer Rechnungsverwaltung erstellen.

Voraussetzung

Wir gehen an dieser Stelle von der Version der Lösung aus, die wir in den drei Beiträgen **Rechnungsverwaltung, Teil 1: Grundlagen** (**www.datenbankentwickler.net/221**), **Rechnungsverwaltung, Teil 2: Rechnungspositionen** (**www.datenbankentwickler.net/222**) und **Rechnungsbericht mit XAML** (**www.datenbankentwickler.net/223**) erstellt haben.

Ziel der Umstellung

Unsere bisherige Version der Rechnungsverwaltung verwendet den SQL Server als Datenbanksoftware. Diesen muss man zunächst installieren, bevor man eine Anwendung nutzen kann, die eine SQL Server-Datenbank zum Speichern ihrer Daten einsetzt. Für viele Anwendungsfälle ist das sinnvoll – vor allem, wenn ohnehin schon eine SQL Server-Installation vorhanden ist. Falls nicht, kann man auch eine der kostenlosen Varianten des SQL Servers installieren.

Wenn man jedoch eine Anwendung programmiert, die man beispielsweise zum Download für Benutzer mit einfachen Anforderungen anbieten möchte, ist der Aufwand zum Installieren der SQL Server-Software übertrieben hoch. Der Benutzer möchte die Anwendung herunterladen, in wenigen Sekunden bis Minuten installieren und diese dann in Betrieb nehmen.

Für diese Zwecke gibt es wesentlich leichtgewichtigere SQL-Datenbanken – zum Beispiel **SQLite**. Um eine **SQLite**-Datenbank zu nutzen, brauchen Sie nur ein paar Dateien im Verzeichnis Ihrer Anwendung zu installieren – und die eigentliche Datenbank kommt ebenfalls in Form einer handlichen Datei. Deshalb möchten wir dafür sorgen, dass Sie die Rechnungsverwaltung genauso mit SQLite nutzen können wie es uns zuvor mit dem SQL Server gelungen ist.

Umstellen auf SQLite

Die Umstellung nehmen wir nun Schritt für Schritt vor. Um Probleme bezüglich der Kompatibilität zwischen den einzelnen Komponenten zu vermeiden, entfernen wir zunächst alle per NuGet-Paketmanager hinzugefügten Pakete. Dazu öffnen wir den Paket-Manager über den Kontextmenü-Eintrag **NuGet-Pakete verwalten...** des Projekt-Elements im Projektmappen-Explorer.

Hier wechseln Sie, wenn dieser nicht bereits angezeigt wird, zum Bereich **Installiert** und entfernen den einzigen Eintrag **Entity-Framework**. Dazu klicken Sie diesen Eintrag an und betätigen dann die **Deinstallieren**-Schaltfläche (siehe Bild 1).

Hinzufügen der benötigten Pakete

Nun wechseln Sie zum Bereich **Durchsuchen** und geben den folgenden Suchbegriff ein:

```
System.Data.SQLite.EF6.
Migrations
```

Dieses Paket erscheint dann in der Ergebnisliste ganz oben. Markieren Sie diesen Eintrag und klicken Sie rechts auf die Schaltfläche **Installieren** (siehe Bild 2).

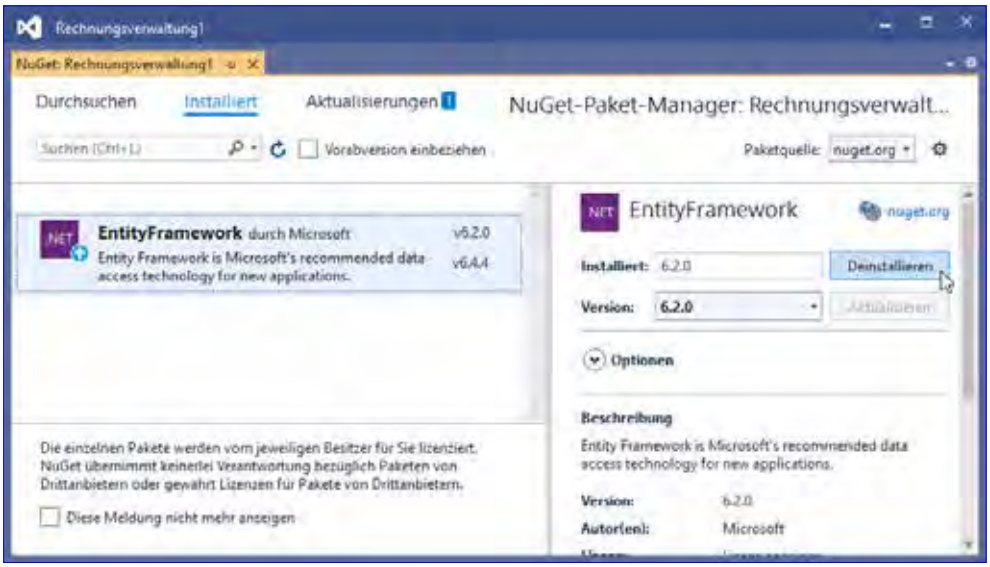


Bild 1: Entfernen des aktuellen **EntityFramework**-Pakets

Der Vorgang dauert einige Sekunden, weil wir hier das unterste Element in der Hierarchie voneinander abhängiger Pakete ausgewählt haben. Wohlwissend, dass die benötigten Elemente, die sich in der Hierarchie über diesem Paket befinden, automatisch mit installiert werden. Zwischenzeitlich akzeptieren Sie die Lizenz für das Element **Apache-2.0**. Danach werden weitere Pakete installiert.

Es kann sein, dass nun einige Fehlermeldungen im Bereich **Fehlerliste** erscheinen. Wenn Sie die Anwendung nun einmal neu erstellen (Menüeintrag **Erstellen|Projektmappe erstellen**) sollten diese jedoch verschwinden. Wechseln Sie nun unter **NuGet** zum Bereich **Installiert** und leeren Sie das Suchfeld, finden Sie die Einträge aus Bild 3 vor.

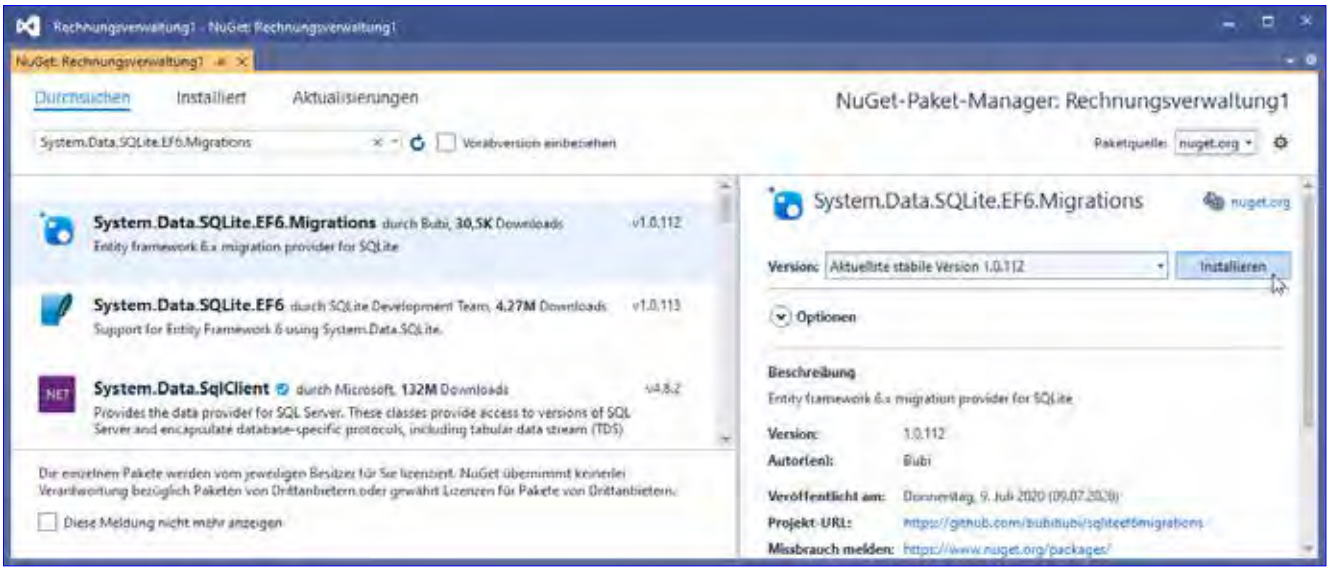


Bild 2: Hinzufügen des Pakets **System.Data.SQLite.EF6.Migrations**

NuGet hat also alle abhängigen Pakete, die für den Einsatz des Pakets **System.Data.SQLite.EF6.Migrations** nötig sind, automatisch hinzugefügt.

Aber warum haben wir überhaupt dieses Paket ausgewählt? Weil bei unseren Experimenten genau dieses Paket fehlt, um den entscheiden Baustein zum Projekt hinzuzufügen – mehr dazu weiter unten.

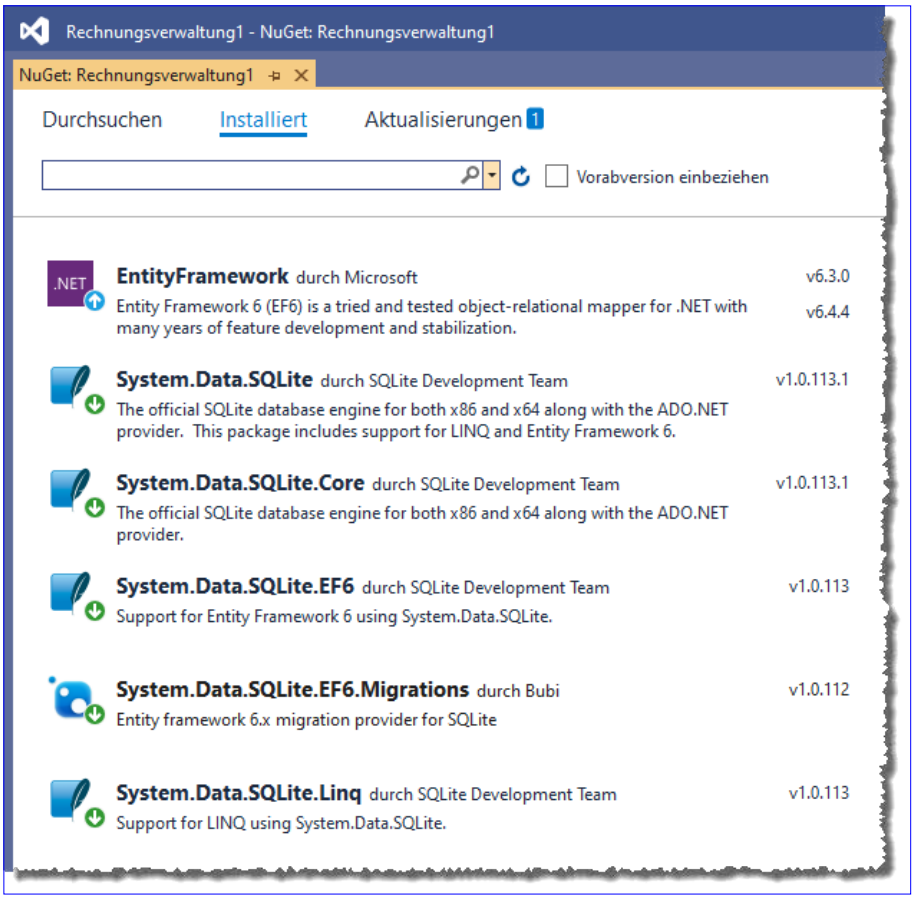


Bild 3: Automatisch mit installierte abhängige Pakete

Änderungen in der Konfigurationsdatei

Wenn Sie sich nun die Datei **App.config** ansehen, die Anwendungseinstellungen im XML-Format bereithält, finden Sie dort zwar noch einige Elemente der unter SQL Server lauffähigen Version der Rechnungsverwaltung, aber durch die Installation der NuGet-Pakete wurden auch einige neue Elemente hinzugefügt.

Allerdings befindet sich dort noch die alte Verbindungzeichenfolge:

```
<connectionStrings>
  <add name="RechnungsverwaltungContext" connectionString="data source=(LocalDb)\MSSQLLocalDB;initial
catalog=Rechnungsverwaltung1.RechnungsverwaltungContext;integrated security=True;MultipleActiveResultSets=True;App=Entity
Framework" providerName="System.Data.SqlClient" />
</connectionStrings>
```

Diese ersetzen wir als Erstes durch die neu zu verwendende Verbindungszeichenfolge. Wichtig ist hier, dass Sie den Namen der Context-Klasse für das Attribut **Name** angeben. Für das Attribut **connectionString** brauchen Sie nur den Pfad zu der zu erstellenden Datei anzugeben – wir verwenden hier **Rechnungsverwaltung.sqlite**:

```
<connectionStrings>
  <add name="RechnungsverwaltungContext" connectionString="Data Source=.\Rechnungsverwaltung.sqlite"
      providerName="System.Data.SQLite" />
</connectionStrings>
```

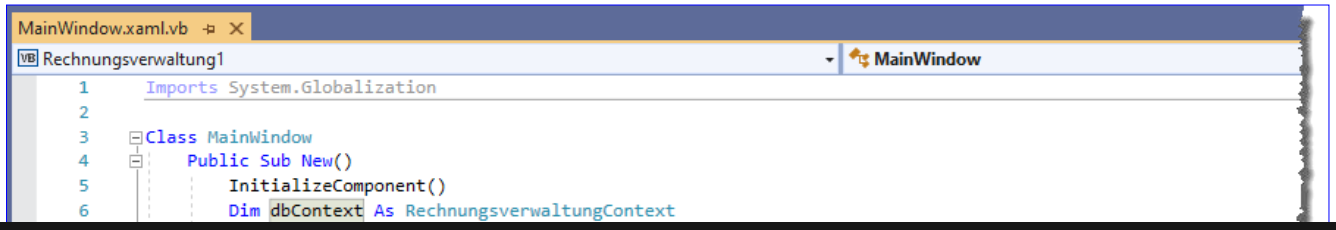
Den Bereich **configurationIentityFramework** erweitern wir dann noch wie folgt um einen Eintrag für **System.Data.SQLite**:

```
<entityFramework>
  <providers>
    <provider invariantName="System.Data.SqlClient" type="System.Data.Entity.SqlServer.SqlProviderServices,
        EntityFramework.SqlServer" />
    <provider invariantName="System.Data.SQLite.EF6" type="System.Data.SQLite.EF6.SQLiteProviderServices,
        System.Data.SQLite.EF6" />
    <provider invariantName="System.Data.SQLite" type="System.Data.SQLite.EF6.SQLiteProviderServices,
        System.Data.SQLite.EF6" />
  </providers>
</entityFramework>
```

Inkompatible Datentypen anpassen

Wenn wir die Anwendung nun starten, erhalten wir die Fehlermeldung aus Bild 4. Dieser verweist darauf, dass kein Datentyp namens **datetime2** unter SQLite gefunden werden konnte. Genau diesen definieren wir aber in der Datei **Rechnung.vb** für das Feld **Rechnungsdatum**:

```
<Column(TypeName:="datetime2")>
Public Property Rechnungsdatum As System.DateTime
...
```



Die Leseprobe dieses Artikels ist hier zu Ende.

Willst Du mehr?

ZUM SHOP

2.400 Seiten Know-how rund um die Datenbankentwicklung mit .NET

André Minhorst Verlag

Spare 20 EUR mit Code dbew20

AUSGABE 04-05/2020

DATENBANK ENTWICKLER

MAGAZIN FÜR DIE DATENBANKENTWICKLUNG MIT VISUAL STUDIO FÜR DESKTOP, WEB UND CO.

TOP-THEMEN:

INTERAKTIV	Lesen und Schreiben von XML-Dokumenten	SEITE 51
VB-GRUNDLAGEN	Fehlerbehandlung unter VB.NET	SEITE 24
VISUAL STUDIO	Debugging in Visual Studio	SEITE 11
LÖSUNGEN	Onlinebanking mit DDBAC: Benutzeroberfläche	SEITE 107

André Minhorst Verlag

Entity Framework: Aktionsabfragen mit LINQ

Wer lange mit Access gearbeitet hat, schreibt Aktionsabfragen entweder mit der Entwurfsansicht für Abfragen oder erstellt diese per SQL-Quellcode. Vielleicht kombinieren Sie auch beide Varianten und passen per Entwurf erstellte Abfragen in der SQL-Ansicht an. Unter VBA verwendet man außerdem oft DAO, um Datensatzgruppen zu öffnen und Daten anzulegen, zu bearbeiten oder zu löschen. In einem Entity Data Model können Sie auch SQL-Abfragen absetzen, sowohl für Auswahl- als auch für Aktionsabfragen. Allerdings gibt es auch LINQ – die Abfragesprache für das Entity Framework. Dieser Artikel zeigt, wie Sie gängige SQL-Aktionsabfragen oder DAO-Aktionen in LINQ übersetzen.

Beispieldateien

Im Download zu diesem Artikel finden Sie einen Ordner mit den Laufzeitdateien der als Beispiel verwendeten Rechnungsverwaltung. Außerdem finden Sie in der Datei **VonSQLZuLINQ.linq** alle in diesem Artikel vorgestellten Beispiele.

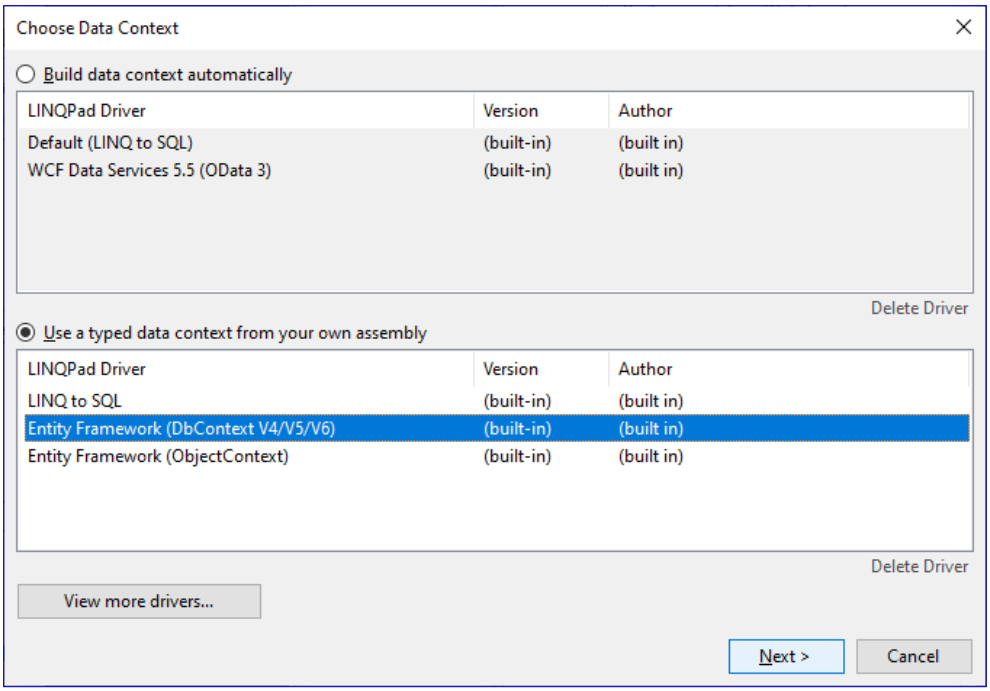


Bild 1: Einrichten einer Datenquelle mit LINQ

Was ist LINQ?

LINQ heißt **Language Integrated Query** und ist die Abfragesprache für Entity Data Models. Im Gegensatz zu VBA, wo Sie die Abfragen immer in SQL formulieren und als Parameter von Methoden wie **OpenRecordset** (bei Auswahlabfragen) oder **Execute** (bei Aktionsabfragen) angegeben haben, bietet LINQ die direkte Integration in die Sprache Visual Basic (und auch C#).

Im Artikel **EF: Daten abfragen mit VB und LINQ** (**www.datenbankentwickler.net/166**) haben wir bereits gezeigt, wie Sie Auswahlabfragen mit LINQ und VB realisieren. Nun schauen wir uns an, wie wir Aktionsabfragen unter LINQ abbilden können.

IntelliSense mit LINQ nutzen

LINQ bietet gegenüber SQL-Abfragen den Vorteil, IntelliSense nutzen zu können. Am einfachsten testen können wir LINQ-Ausdrücke nach wie vor mit dem Tool **LINQPad**. Wir arbeiten hier mit der Version 5. Um auf die Daten einer SQL Server-Datenbank zugreifen zu können, richten wir mit folgenden Schritten eine Verbindung ein.

Die einzige Voraussetzung ist, dass es eine **.exe** für das Projekt gibt und dass Sie wissen, wo die Konfigurationsdatei gespeichert ist – was nach dem Debuggen eines Desktop-Projekts mit Visual Studio aber beides vorhanden ist:

- Starten Sie LINQPad 5.
- Klicken Sie im linken Bereich auf **Add connection**.
- Wechseln Sie im Dialog **Choose Datacontext** aus Bild 1 zum unteren Bereich und wählen Sie dort den Eintrag **Entity Framework (DbContext V4/V5/V6)** aus und klicken Sie auf **Next**.
- Wählen Sie im Dialog **Entity Framework dbContext Connection** die Dateien für die Eigenschaften **Path to Custom Assembly** und **Path to application config file** aus. Hier wählen Sie die **.exe**-Datei aus, beispielsweise aus dem Ordner **Rechnungsverwaltung** aus dem Download zu diesem Artikel. Nach der Auswahl der ersten Datei werden Sie in einem weiteren Dialog aufgefordert, die Datenbank-Context-Klasse auszuwählen.
- Anschließend sieht der Dialog **Entity Framework DbContext Connection** wie in Bild 2 aus. Hier können Sie mit der Schaltfläche **Test** noch prüfen, ob die Verbindung hergestellt werden kann. Mit dem Aktivieren der Option **Remember this connection** stellen Sie sicher, dass die Verbindung auch beim nächsten Start von **LINQPad** noch vorhanden ist.

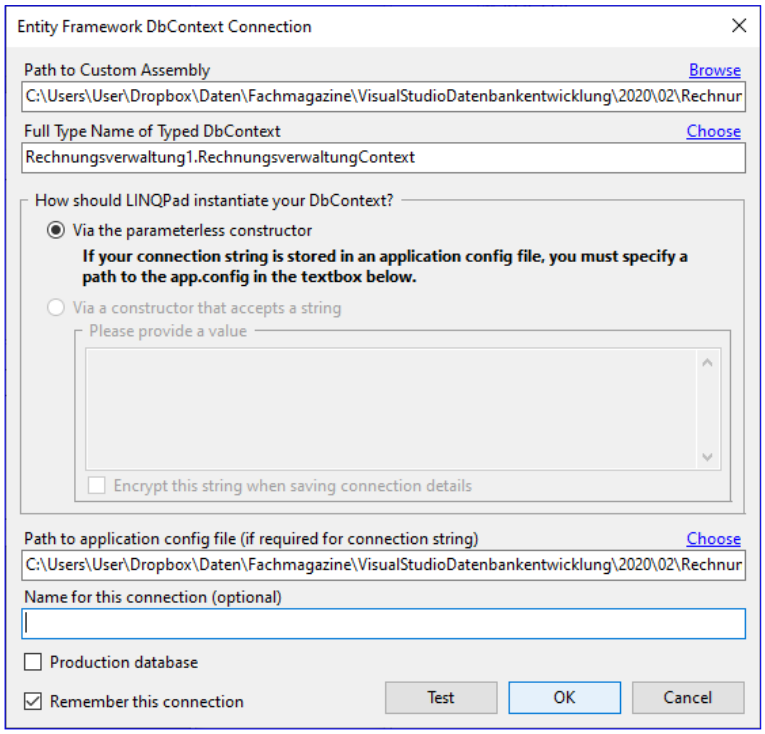


Bild 2: Einrichten der Verbindungsinformationen

Der linke Bereich zeigt nun die Entitäten und Felder der Verbindung an (siehe Bild 3). Mit einem Klick der rechten Maustaste auf eine der Entitäten öffnen Sie ein Kontextmenü, das bereits einige mögliche Befehle für diese Entität anzeigt. Klicken Sie einen dieser Einträge an, wird die passende LINQ-Abfrage im Query-Fenster eingefügt.

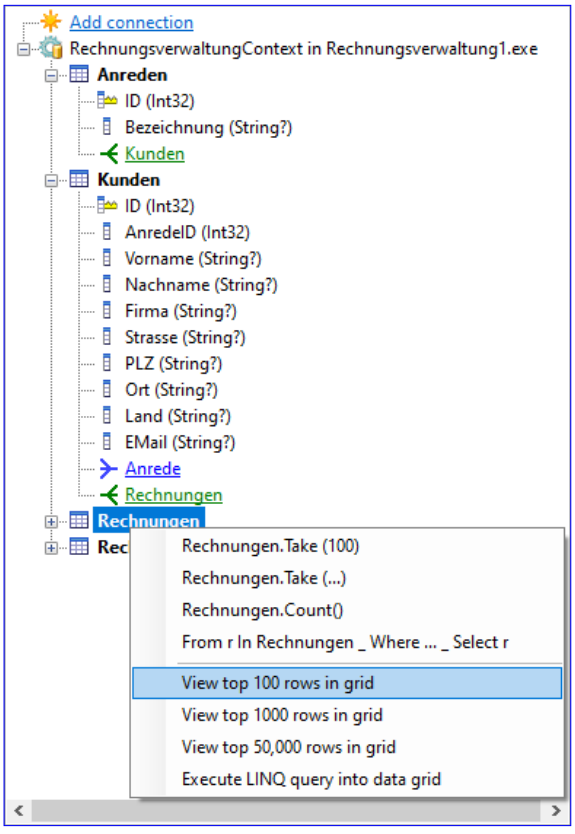


Bild 3: Die Entitäten der neuen Verbindung

Standardmäßig zeigt dies nun eine C#-Abfrage an. Wenn Sie möchten, dass automatisch VB-Abfragen erstellt werden, öffnen Sie

mit dem Menüeintrag **Edit|Preferences** den Optionen-Dialog und stellen Sie auf der Seite **Query** die Option **Default Query Language** auf **VBA Expression** oder **VB Statement(s)** ein.

Damit sind Sie nun auch bereit, eigene LINQ-Abfragen auf Basis der Entitäten des Entity Data Models abzusetzen. Diese geben Sie einfach auf einer Seite mit dem Wert **VB Expression** im Auswahlfeld **Language** ein. Sie können übrigens in LINQPad den Verweis auf die Context-Klasse des Entity Data Models weglassen.

Ein Zugriff auf die Entitätslisten kann auch ohne Angabe dieser Klasse erfolgen. Allerdings ist das in Visual Studio nicht möglich, sodass wir in den folgenden Beispielen jeweils auf das Objekt **dbContext** zugreifen. Die Deklaration und Initialisierung wollen wir jedoch nur einmal anführen:

```
Dim dbContext As RechnungsverwaltungContext
dbContext = New RechnungsverwaltungContext
```

Einen Datensatz hinzufügen

Um einen Datensatz hinzuzufügen, benötigen wir noch nicht einmal einen LINQ-Ausdruck. Wir erstellen einfach ein neues Objekt des gewünschten Typs, hier **Kunde**, und weisen diesem die Eigenschaftswerte zu. Dann fügen wir das neue **Kunde**-Objekt mit der **Add**-Methode zur Auflistung **Kunden** hinzu und übernehmen die Änderungen durch einen Aufruf der **SaveChanges**-Methode in die Datenbank. Schließlich geben wir als Nachweis des erfolgreichen Anlegens noch den Wert des Primärschlüsselfeldes des neuen Datensatzes aus:

```
'###Beispiel 1
'Einen Kunden hinzufügen
Dim dbContext As RechnungsverwaltungContext
dbContext = New RechnungsverwaltungContext
Dim NeuerKunde As Kunde
NeuerKunde = New Kunde
With NeuerKunde
    .AnredeID = 1
    .Vorname = "André"
    .Nachname = "Minhorst"
End With
dbContext.Kunden.Add(NeuerKunde)
dbContext.SaveChanges
Debug.Print(NeuerKunde.ID)
```

Platzsparende Schreibweise für das Anlegen eines neuen Elements

Wir können ein paar Zeilen Code sparen, wenn wir das Anlegen wie folgt programmieren:

```
'###Beispiel 2
'Kunde hinzufügen, Kurzfassung
```

```
Dim NeuerKunde As Kunde
NeuerKunde = New Kunde With {.AnredeID = 1, .Vorname = "André", .Nachname = "Minhorst"}
dbContext.Kunden.Add(NeuerKunde)
dbContext.SaveChanges
Debug.Print(NeuerKunde.ID)
```

Die **NeuerKunde**-Variable könnten wir auch noch einsparen, wenn wir später nicht die **ID** des neuen Kunden ausgeben wollten:

```
dbContext.Kunden.Add(New Kunde With {.AnredeID = 1, .Vorname = "André", .Nachname = "Minhorst"})
dbContext.SaveChanges
```

Mehrere Datensätze hinzufügen

Wenn Sie mehrere Datensätze hinzufügen möchten, können Sie dies natürlich erledigen, indem Sie einfach immer einen neuen Kunden erstellen, seine Eigenschaften füllen und diesen mit der **Add**-Methode dem jeweiligen **DbSet**-Element zuweisen.

Dazu erstellen wir zuerst eine Liste mit Elementen des Typs **Kunde** und fügen dieser dann mit der **Add**-Methode die neuen **Kunde**-Elemente hinzu – mit der kurzen Schreibweise, die wir zuvor vorgestellt haben:

```
'### Beispiel 3
'Mehrere Kunden hinzufügen
Dim dbContext As RechnungsverwaltungContext
dbContext = New RechnungsverwaltungContext
Dim NeueKunden As List(Of Kunde)
NeueKunden = New List(Of Kunde)
NeueKunden.Add(New Kunde With {.AnredeID = 1, .Vorname = "André", .Nachname = "Minhorst"})
NeueKunden.Add(New Kunde With {.AnredeID = 2, .Vorname = "Claudia", .Nachname = "Müller"})
```

Danach geben wir die aktuelle Anzahl von Kunden in der Datenbank aus und rufen dann die **AddRange**-Methode auf, der wir die Liste der Kunden namens **NeueKunden** als Parameter übergeben. Nach dem Speichern der Änderungen mit der **SaveChanges**-Methode geben wir die neue Anzahl Kunden aus:

```
Debug.Print("Vorher: " & dbContext.Kunden.Count.ToString)
dbContext.Kunden.AddRange(NeueKunden)
dbContext.SaveChanges
Debug.Print("Nachher: " & dbContext.Kunden.Count.ToString)
```

Verknüpfte Daten hinzufügen

Wenn Sie beispielsweise einen **Rechnung**-Datensatz anlegen und diesen gleich um einen oder mehrere Rechnungsposition-Datensätze ergänzen wollen, gehen Sie wie folgt vor:

```
'### Beispiel 4
```

Debugging in Visual Studio

Wenn eine Anwendung sich nicht so verhält wie gewünscht oder diese sogar Ausnahmen auslöst, wollen wir herausfinden, warum das so ist. Wurde ein Wert nicht oder falsch gesetzt, konnte ein Objekt nicht gefüllt werden oder was ist die Ursache für das Problem? Hier kommen die Debugging-Techniken von Visual Studio ins Spiel. Der vorliegende Artikel stellt die gängigsten Techniken zum Debuggen von Code unter Visual Studio vor.

Das Wort **Debuggen** kommt der Sage nach vom Wort Bug. Früher wurden zum Durchführen von Rechenoperationen in Computern Relais verwendet, die durch Insekten blockiert werden konnten. Das Entfernen dieser Tierchen wurde schließlich **Debuggen** genannt – ein Begriff, der sich bis heute gehalten hat. Nur, dass Bug heute nicht mehr sprichwörtlich Käfer bedeutet, sondern schlicht Programmierfehler.

Anwendung zum Debuggen starten

Grundsätzlich gibt es zwei Modi zum Ausführen von .NET-Anwendungen: Den Debug-Modus und den Release-Modus. Der Release-Modus erzeugt eine leistungsoptimierte Version der Anwendung – diese entspricht auch der Version der Anwendung, die Sie mit dem Erstellen eines Projekts erhalten. Der Debug-Modus hingegen erlaubt es, bei Fehlern den Quellcode anzuzeigen, der für den Fehler verantwortlich ist. Außerdem können Sie in diesem Modus Informationen über den Zustand von Objekten und Variablen erhalten, die gerade im Gültigkeitsbereich liegen und Meldungen ausgeben sowie weitere Funktionen nutzen, die wir Ihnen im Laufe dieses Artikels vorstellen werden. Standardmäßig ist für das Starten einer Anwendung von Visual Studio aus der Debug-Modus eingestellt. Diese Einstellung können Sie leicht über das Menü von Visual Studio ändern (siehe Bild 1).

Um die Anwendung nun im Debugging-Modus zu starten, betätigen Sie entweder die Taste **F5**, klicken auf die Schaltfläche mit dem grünen Pfeil und dem Text **Starten** oder wählen den Menü-Eintrag **Debuggen|Debuggen starten**.

Zum Debuggen anhalten

Um eine Anwendung zu debuggen, muss der laufende Code angehalten werden. Dazu gibt es verschiedene Möglichkeiten:

- Es tritt eine Ausnahme auf, also ein Fehler im Code, der eine weitere Ausführung des Codes verhindert. Ein Beispiel sehen Sie in Bild 2.

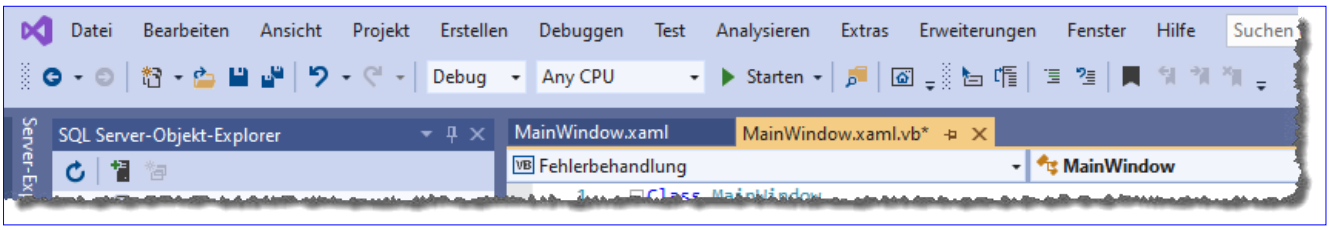


Bild 1: Einstellen des Ausführungsmodus, hier **Debug**

- Sie legen einen Haltepunkt im Code fest, damit der Code auch ohne einen Laufzeitfehler an der gewünschten Stelle angehalten und untersucht werden kann.

- Sie fügen der Anwendung an der gewünschten Stelle die Methode **Break** der Klasse **Debugger** hinzu. Dies entspricht der **Stop**-Anweisung, die Sie vielleicht von Access/VBA kennen. Der Code wird genau an dieser Stelle angehalten.

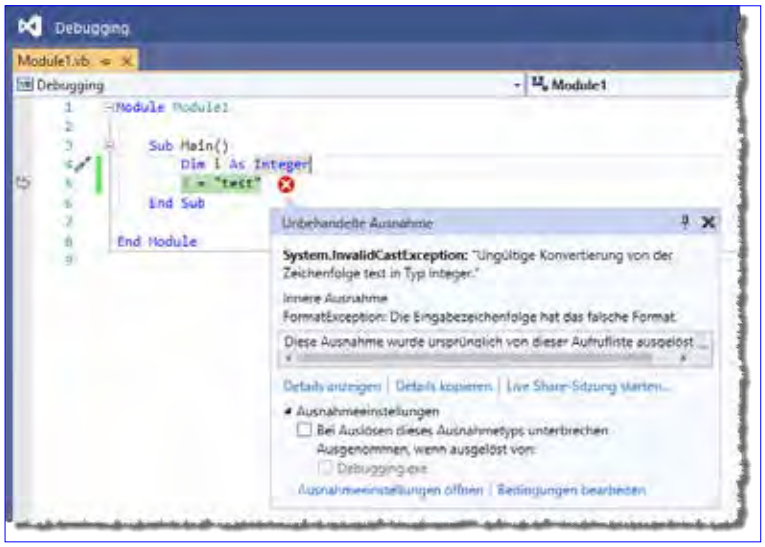


Bild 2: Debuggen wird durch Laufzeitfehler ermöglicht

Haltepunkt setzen

Einen Haltepunkt im Code setzen Sie, indem Sie im Codefenster in der Zeile, in der die Ausführung angehalten werden soll, in den linken, grauen Bereich klicken, sodass dort ein roter Punkt erscheint (siehe Bild 3).

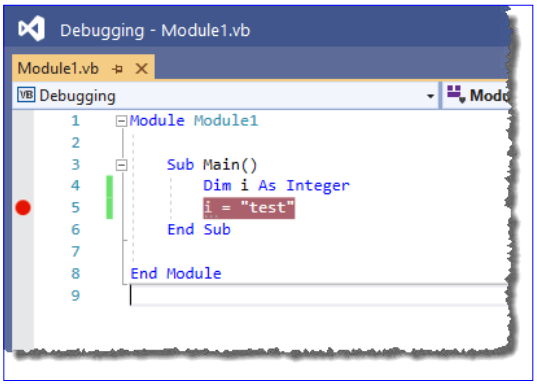


Bild 3: Haltepunkt

Eine alternative Möglichkeit zum Hinzufügen von Haltepunkten ist die Taste **F9**. Bewegen Sie die Einfügemarke in die gewünschte Zeile und betätigen Sie diese Taste, um einen Haltepunkt hinzuzufügen oder diesen wieder zu entfernen.

Sie können alle ausführbaren Zeilen mit einem Haltepunkt versehen, was alle Zeilen mit Ausnahme der Deklarationszeilen umfasst.

Wenn die Ausführung an der markierten Zeile stoppt, bedeutet das übrigens, dass die markierte Zeile noch nicht ausgeführt wurde. Wenn Sie also Variablen in einer Zeile nach der Ausführung (etwa der Zuweisung eines Wertes an eine Variable) untersuchen möchten, können Sie den Haltepunkt auch gleich auf die folgende Zeile verschieben.

Haltepunkte per Code

Eine weitere Möglichkeit, einen Haltepunkt zu setzen, bietet die Klasse Debugger. Diese stellt die Methode **Break** zur Verfügung, die beim Erreichen den Code unterbricht (siehe Bild 4).

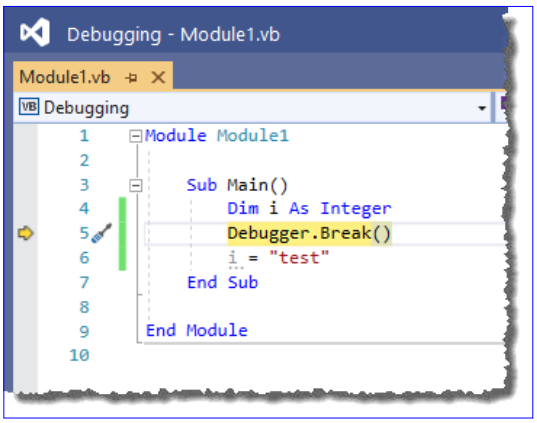


Bild 4: Programmierter Stop

Warum sollte man diese Methode nutzen, statt einfach einen Haltepunkt zu setzen? Unter Access und dem VBA-Editor wurden Haltepunkte gelöscht, wenn man die Anwendung geschlossen und erneut geöffnet hat. Das war unter Access der Grund, die **Stop**-Anweisung zu verwenden – wenn die Anwendung abgestürzt ist, war diese nach dem Neustart wieder vorhanden, die Haltepunkte aber nicht. Unter Visual Studio sind die Haltepunkte allerdings nach dem

Schließen und erneuten Öffnen des Projekts noch vorhanden – eigentlich braucht man **Debugger.Break** also nicht.

Beim Erreichen eines Haltepunktes

Wenn Sie beim Debuggen einen Haltepunkt erreichen, können Sie verschiedene Aktionen durchführen. Die erste ist, Informationen über die aktuelle Situation zu ermitteln. Dazu können Sie beispielsweise über den Namen einer Variablen fahren, um ihren aktuellen Wert zu ermitteln (siehe Bild 5).

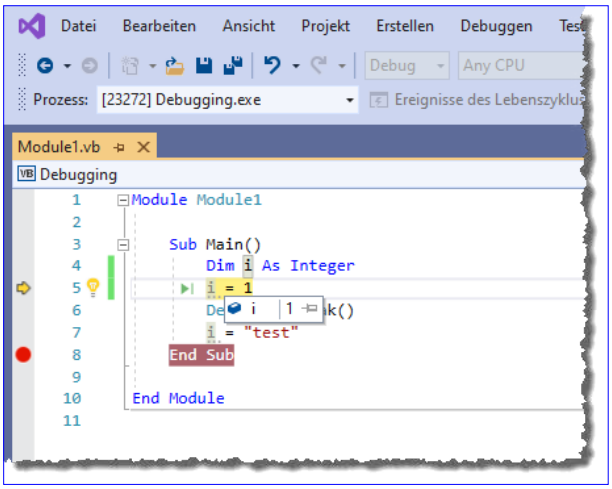


Bild 5: Auslesen von Variableninhalten

Weiter im Code

Wenn Sie einen Haltepunkt erreicht haben und das Debuggen nun fortsetzen wollen, gibt es dazu folgende Möglichkeiten:

- Sie betätigen erneut die Taste **F5** und der Code läuft weiter bis zum Schluss oder bis er durch einen weiteren Haltepunkt oder einen Fehler angehalten wird.

- Sie betätigen die Taste **F11**, um jeweils eine Zeile auszuführen.

Letztere Möglichkeit ist interessant, wenn Sie im Detail die einzelnen Zeilen des Codes durchlaufen wollen.

Laufzeitfehler untersuchen

Wenn Sie beim Debuggen auf einen Laufzeitfehler stoßen, der nicht behandelt wird, können Sie damit einige Informationen erhalten, die für eine Behandlung des Fehlers im Code nützlich sind.

Wie die Fehlermeldung einer unbehandelten Ausnahme aussieht, haben wir bereits weiter oben betrachtet. Hier finden wir noch einige weiterführende Möglichkeiten. Die wichtigste ist der Link **Details anzeigen**, mit dem Sie ein Fenster wie in Bild 6 öffnen können.

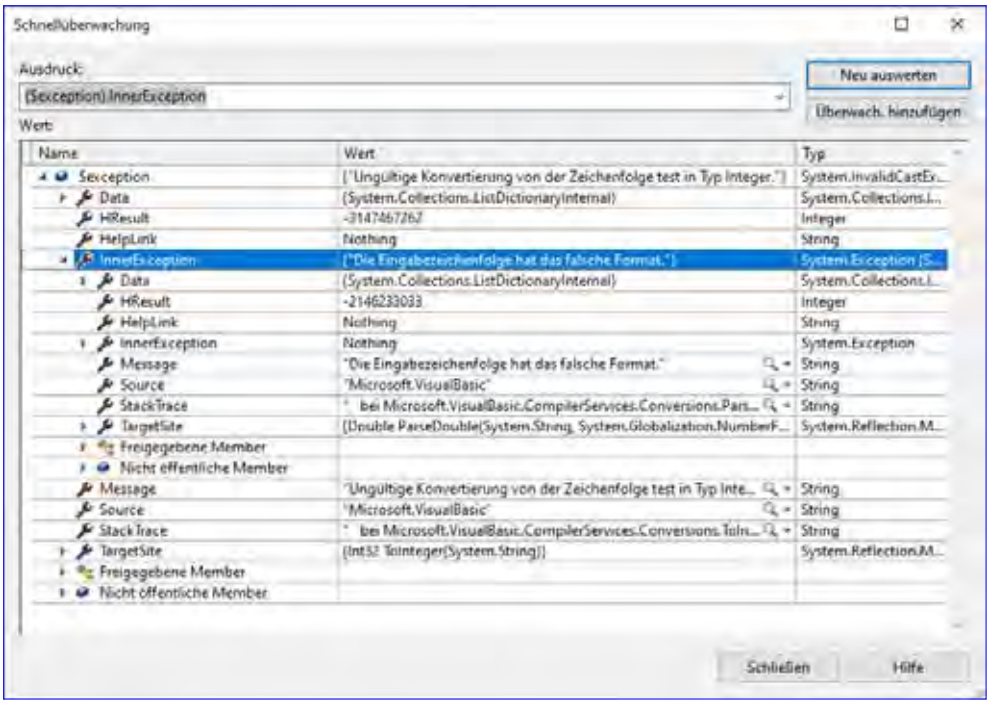


Bild 6: Fehlerdetails beim Auftreten einer Ausnahme

Hier finden Sie detaillierte Informationen zu der Ausnahme. In vielen Fällen können Sie unter **InnerException** noch weitere Informationen anzeigen, wie es in der Abbildung der Fall ist.

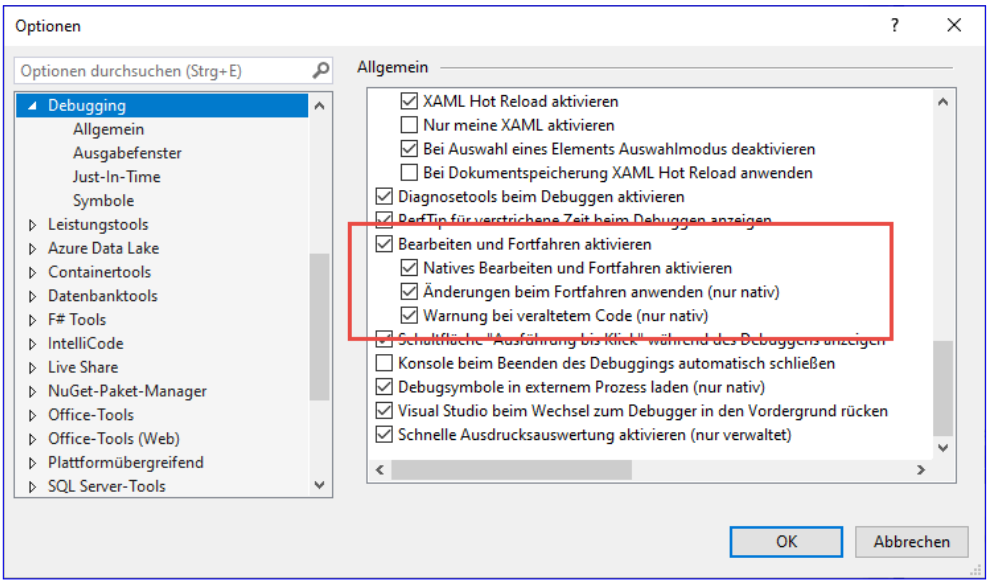


Bild 8: Aktivieren des Bearbeitens von Code während des Debuggens

Code während des Debugging bearbeiten

Eine wichtige Funktion ist das Bearbeiten des Codes während des Debuggings. Damit dies funktioniert, aktivieren Sie die Optionen aus Bild 8 im Bereich **Debugging** des Optionen-Dialogs, den Sie mit dem Menübefehl **Extras|Optionen** öffnen – und die normalerweise bereits aktiviert sind..

Es werden allerdings nicht alle Änderungen unterstützt. Unter folgendem Link finden Sie eine Liste der unterstützten und nicht unterstützten Änderungen:

`https://github.com/dotnet/roslyn/blob/master/docs/wiki/EnC-Supported-Edits.md`

Debugging steuern

Neben dem Starten und Fortsetzen des Debuggings mit **F5** und dem Durchführen von Einzelschritten mit **F11** gibt es noch weitere Optionen, die Sie auch im Menüpunkt **Debuggen** finden (siehe Bild 7). Wenn Sie einmal die Tastenkombination für einen der Befehle suchen, können Sie diese auch in diesem Menü einsehen. Die Befehle im Einzelnen:

- **Weiter** (**F5**): Setzt den Code bis zum nächsten Haltepunkt oder Fehler beziehungsweise bis zum Ende des Codes durch.
- **Alle unterbrechen** (**Strg + Alt + Pause**): Unterbricht an der aktuell ausgeführten Codezeile.
- **Debuggen beenden** (**Umschalt + F5**): Beendet das Debuggen.

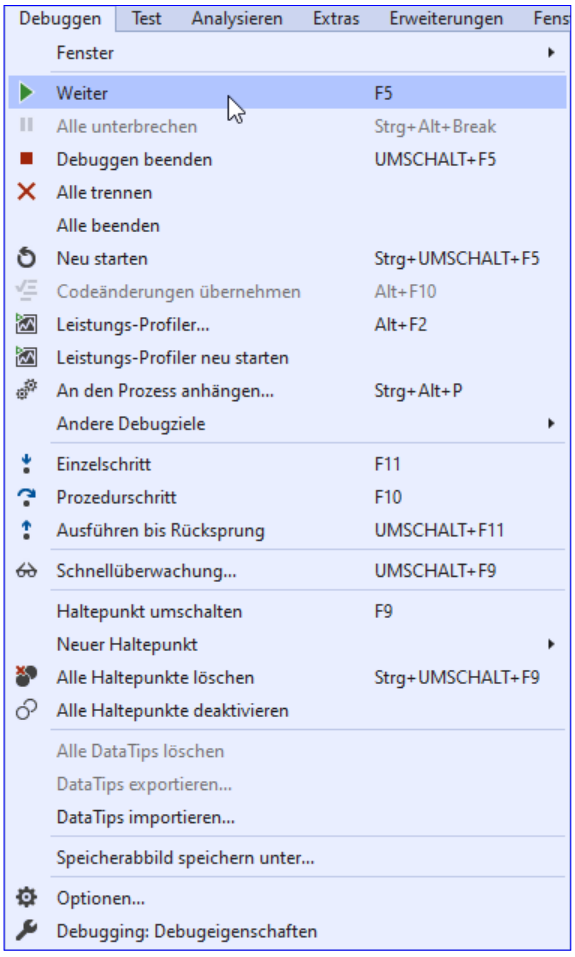


Bild 7: Kontextmenü-Befehle zum Debuggen

- **Neu starten** (**Strg + Umschalt + F5**): Beendet das Debugging und startet es erneut.

- **Prozedurschritt** (**F10**): Führt die Anweisungen der aktuellen Methode im Einzelschrittmodus aus, aber springt nicht in andere, von dieser Methode aus aufgerufene Methoden.

- **Ausführen bis Rücksprung** (**Umschalt + F11**): Springt von der aktuellen Methode direkt zur aufrufenden Methode, wenn überhaupt ein Aufruf von einer anderen Methode erfolgt ist.

Damit erhalten wir recht exakt die gleichen Funktionen und Tastenkombinationen wie im VBA-Editor von Access – mit dem Unterschied, dass statt der Taste **F8** die Taste **F11** zum Einsatz kommt.

Es gibt noch weitere Befehle, die allerdings erst bei laufendem Debugging durch einen Mausklick mit der rechten Maustaste in das Codefenster auftauchen (siehe Bild 9). Interessant sind hier die folgenden Befehle:

- **HaltepunktlHaltepunkt einfügen**: Fügt einen Haltepunkt in der aktuellen Zeile ein.

- **Nächste Anweisung anzeigen** (**Alt + Num ***): Durch das Betätigen dieses Kontextmenübefehls beziehungsweise der Tastenkombination aus der **Alt**-Taste und der ***-Taste des Nummernblocks wird die Einfügemarke auf die nächste auszuführende Anweisung verschoben. Dieser Befehl ist sinnvoll, wenn Sie während des Debuggens andere Stellen im Code betrachten und dabei die aktuell auszuführende Anweisung aus dem sichtbaren Bereich verschwindet – Sie gelangen dann schnell wieder zur aktiven Stelle zurückl

- **Ausführen bis Cursor** (**F10**): Mit diesem Befehl können Sie die Einfügemarke an eine beliebige Stelle im Code platzieren und den Code dann bis zu dieser

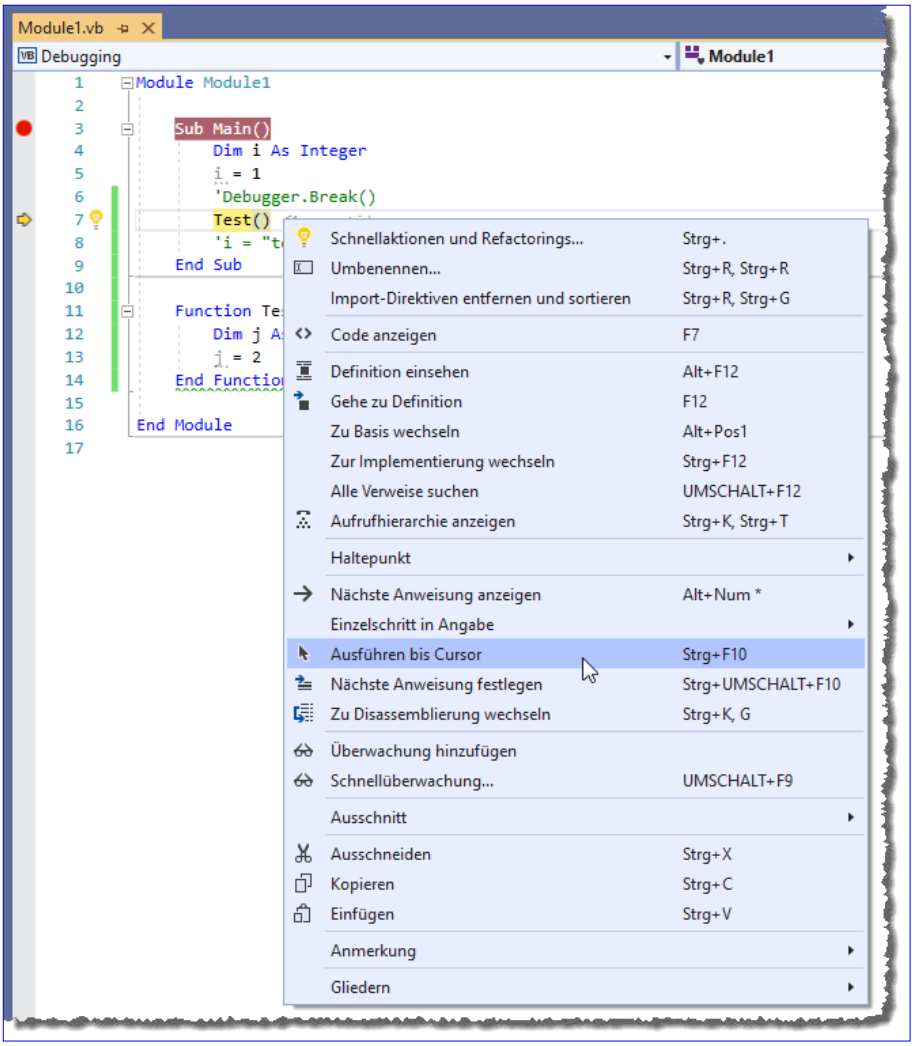


Bild 9: Weitere Debugging-Befehle

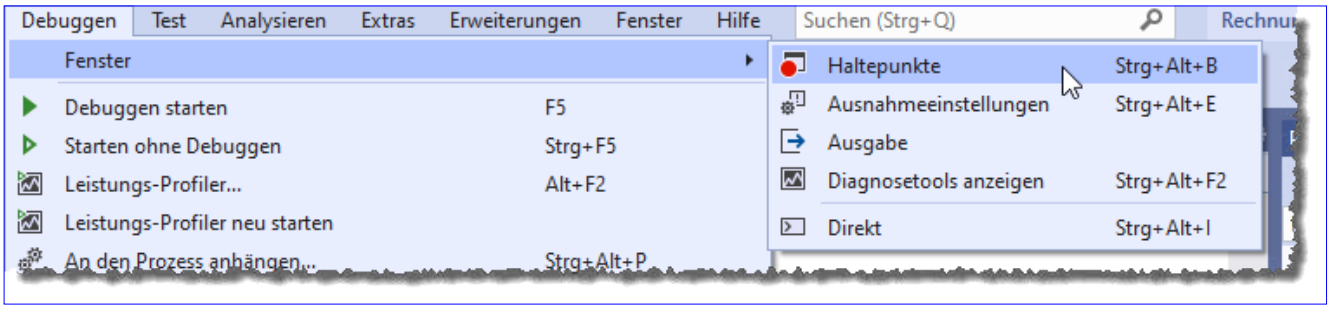


Bild 10: Aktivieren des Tool Windows zur Verwaltung der Haltepunkte

Stelle durchlaufen lassen. Das ist die Kurzversion für das Setzen eines Haltepunkts, der direkt wieder entfernt werden soll.

- **Nächste Anweisung festlegen** (**Strg + Umschalt + F10**): Wenn Sie aktuell schrittweise durch den Code laufen oder dieser anderweitig unterbrochen ist (die aktuelle Zeile ist gelb hinterlegt), können Sie direkt zu einer anderen Zeile springen – und zwar zu der, in der sich bei Betätigen des Menübefehls beziehungsweise der Tastenkombination die Einfügemarke befindet. Dies gelingt jedoch nur für Anweisungen innerhalb der aktuellen Methode.

Tool Windows mit Debugging-Informationen

Visual Studio bietet eine ganze Reihe von Tool Windows, die verschiedene Informationen anzeigen – darunter auch Debugging-Informationen.

Haltepunkte verwalten

In längeren Debugging-Sessions mit Fehlersuche kommen gern einige Haltepunkte zusammen. Diese müssen Sie nicht mühsam von Hand aus den betroffenen Modulen entfernen – Sie können ein Tool Window verwenden, das die Haltepunkte übersichtlich anzeigt und die Möglichkeit bietet, die Haltepunkte zu verwalten. Dieses Tool Window zeigen Sie entweder über den Menüeintrag **Debuggen|Fenster|Haltepunkte** (siehe Bild 10) oder über die Tastenkombination **Strg + Alt + B** an.

Dies aktiviert das Tool Window namens **Haltepunkte** (siehe Bild 11). Die Liste zeigt alle aktuellen Haltepunkte des Projekts an. Der Bereich bietet verschiedene Möglichkeiten. Um alle markierten Haltepunkte schnell loszuwerden, klicken Sie einfach auf die **Löschen**-Schaltfläche. Nicht zu entfernende Haltepunkte können Sie vorher abwählen.

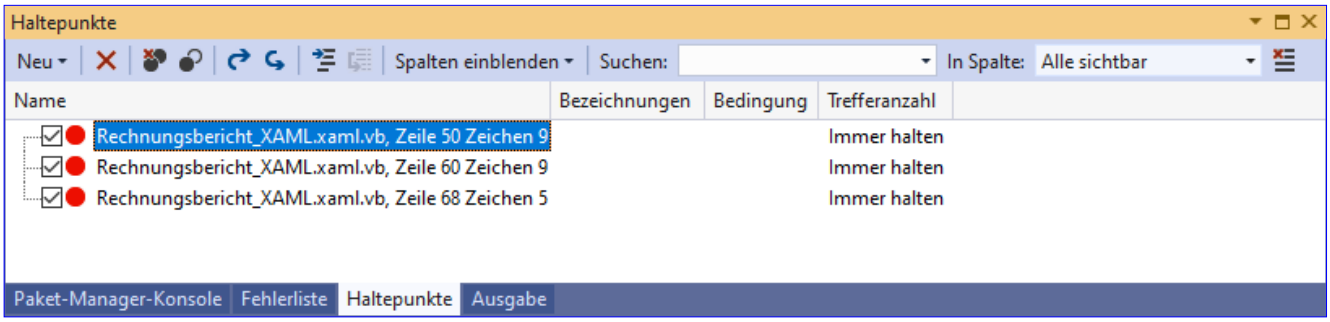


Bild 11: Anzeige der Haltepunkte einer Anwendung

Fehlerbehandlung unter VB.NET

Was wäre eine Anwendung ohne Fehlerbehandlung? Auch wenn wir in den bisherigen Beispielanwendungen meist gar keine Fehlerbehandlung genutzt haben, weil der Code so übersichtlicher ist. Wenn Sie aber eine Anwendung an Kunden weitergeben, sollte diese keine unbehandelten Ausnahmen liefern. Deshalb beschreiben wir im vorliegenden Artikel, wie die Fehlerbehandlung unter VB.NET funktioniert und welche Unterschiede sich zur Fehlerbehandlung unter Access/VBA ergeben.

Beispiele ausprobieren

Wir haben die Beispiele im Tool LINQPad eingegeben und ausprobiert. Das Tool ist in der Basisversion kostenlos und erlaubt das Ausführen von VB.NET-Prozeduren, ohne das Sie immer erst ein Projekt zum Debuggen starten müssen. Den Download des Tools finden Sie unter **www.linqpad.net**.

Von VBA zu VB.NET

Unter VBA war die Fehlerbehandlung recht einfach: Wir haben mit **On Error Resume Next** festgelegt, dass die Fehlerbehandlung für die folgenden Zeilen ausgesetzt wird und haben selbst mit **Err.Number** geprüft, ob ein Fehler aufgetreten ist und entsprechend darauf reagiert. Damit haben wir dann gezielt Fehler behandelt. Oder man hat zu Beginn einer Prozedur mit einer Anweisung wie **On Error Goto Fehler** festgelegt, dass die Prozedur im Falle eines Fehlers zu einer Marke namens **Fehler** springen sollte, wo wir dann eine allgemeine Fehlerbehandlung eingebaut haben, die beispielsweise eine E-Mail mit der Fehlermeldung und weiteren Informationen an den Hersteller der Software geschickt hat.

Wenn Sie eine Anwendung von Access nach VB.NET migrieren und dabei möglichst viel Quellcode übernehmen wollen, gibt es allerdings eine gute Nachricht: Die unter VBA verwendeten Befehle für die Fehlerbehandlung funktionieren auch noch unter VB.NET.

Das folgende klassische Beispiel liefert allerdings nicht den Beweis, dass die Fehlerbehandlung wie unter VBA funktioniert. Dafür benötigen Sie allerdings einen Verweis auf den Namespace **Microsoft.VisualBasic**:

```
Sub DivisionByZero
    On Error Resume Next
    Debug.Print(1 / 0)
    If Not (Err.Number = 0) Then
        Debug.Print("Fehler: " + Err.Number + " " + Err.Description)
    End If
End Sub
```

Das Ergebnis der Division von **1** und **0** wird nämlich einfach wie folgt ausgegeben:

∞

Also lösen wir den Fehler mit der entsprechenden .NET-Anweisung aus, nämlich **ThrowException**:

```
On Error Resume Next
Throw New DivideByZeroException()
'Debug.Print(1 / 0)
If Not (Err.Number = 0) Then
    Debug.Print("Fehler: " & Err.Number & " " & Err.Description)
End If
```

Damit haben wir auch gleich die erste Anweisung der Ausnahmebehandlung unter VB.NET kennengelernt – die Thow-Anweisung dient dazu, Ausnahmen auszulösen. Mehr dazu weiter unten! Übrigens sollten Sie, auch wenn es möglich ist, nicht mehr die von VBA gewohnten Möglichkeiten zur Fehlerbehandlung nutzen.

Fehlerbehandlung unter VB.NET

.NET bietet wesentlich umfassendere Möglichkeiten für die Fehlerbehandlung oder, wie man hier sagt, Ausnahmebehandlung. Die grobe Richtschnur für eine Fehlerbehandlung unter VB.NET ist ein Block ähnlich einer **If...Then...Else**-Bedingung mit den drei Anweisungen **Try**, **Catch** und **Finally**. Die Mindestanforderung sind die beiden Anweisungen **Try**, **Catch** und die **End Try**-Anweisung:

```
Try
    'auszuführender Code, der eine Ausnahme auslösen könnte
Catch ex As Exception
    'Code, der beim Auftreten der Ausnahme ausgeführt werden soll
End Try
```

Das Pendant zu der Fehlerbehandlung von oben mit der Division durch 0 würde also wie folgt aussehen:

```
Sub DivisionByZeroVBNET
    Try
        Throw New DivideByZeroException()
    Catch ex As Exception
        Debug.Print("Fehler: " & ex.Message)
    End Try
End Sub
```

Der Aufbau ist also prinzipiell gar nicht so viel anders als unter VBA. Der Teil zwischen **Catch...** und **End Try** wird nur ausgeführt, wenn eine Ausnahme ausgelöst wird. Ein wichtiger Unterschied: Es gibt unter VB.NET kein **Err**-Objekt (zumindest nicht, wenn Sie **Try...Catch** verwenden). Dafür legen Sie in der **Catch**-Anweisung jedoch ein Objekt des Typs **Exception** fest, hier mit der Bezeichnung **ex**. **Exception** liefert uns wie **Err** zwar eine Meldung, aber keine Fehlernummer.

Wie sollen wir das nun vernünftig auswerten und je nach Fehler entsprechend reagieren?

Ganz einfach: Es gibt für jede Ausnahme eine spezielle Exception. Für diesen Fall ist diese leicht zu finden, denn die Exception wird durch das »Werfen« der Ausnahme **DivideByZeroException** ausgelöst. Dementsprechend können wir die **Catch**-Anweisung direkt etwas genauer definieren, sodass diese nur auf **DivideByZeroException**-Ausnahmen reagiert:

```
Sub DivisionByZeroVBNET_DivideByZeroException
    Try
        Throw New DivideByZeroException()
    Catch ex As DivideByZeroException
        Debug.Print("Fehler: " & ex.Message & " ")
    End Try
End Sub
```

Da Sie wissen, welcher Fehler hier behandelt wird, können Sie die Meldung auch anpassen, ohne auf **ex.Message** zuzugreifen:

```
Debug.Print ("Division durch Null.")
```

Tritt nach der **Try**-Anweisung eine andere Ausnahme als **DivideByZeroException** auf, führt dies allerdings zu einer unbehandelten Ausnahme – zum Beispiel wenn Sie mit folgender Anweisung einen Dateifehler auslösen:

```
Throw New FileNotFoundException
```

Wir sollten also auch mit der allgemeinen Ausnahme **Exception** alle möglichen Ausnahmen abfangen. Dazu fügen wir einfach einen weiteren **Catch**-Zweig zu dem Konstrukt hinzu:

```
Try
    Throw New FileNotFoundException
    Throw New DivideByZeroException()
Catch ex As DivideByZeroException
    Debug.Print("Die Datei konnte nicht gefunden werden.")
Catch ex As Exception
    Debug.Print("Fehler: " & ex.Message)
End Try
```

Restarbeiten mit und ohne Ausnahme

Unter VBA haben wir für Restarbeiten, die in jedem Fall, also auch beim Auftreten eines Fehlers zu erledigen sind, wie folgt untergebracht:

```
    On Error Resume Fehler
    'Fehlerauslösende Zeile
Ende:
    'Restarbeiten
```

```
    Exit Sub
Fehler:
    'Fehlerbehandlung
    Goto Ende
```

Wir haben also unter der Marke **Ende** Befehle untergebracht, die auch beim Ausbleiben eines Fehlers ausgelöst werden. Damit die **Fehler**-Marke nur erreicht wird, wenn tatsächlich ein Fehler auftritt, haben wir dieser die **Exit Sub**- beziehungsweise **Exit Function**-Anweisung vorangestellt. Damit die Aufräumarbeiten aber auch nach Auftreten eines Fehlers noch ausgeführt werden, haben wir am Ende der Fehlerbehandlung noch einen Verweis zur Marke **Ende** hinzugefügt.

Unter VB.NET benötigen wir dazu nur einen weiteren Zweig im **Try...End Try**-Konstukt. Dieser Zweig heißt **Finally** und wird immer am Ende eingebaut:

```
Sub TryCatchFinally
    Try
        Throw New FileNotFoundException
        Throw New DivideByZeroException()
    Catch ex As DivideByZeroException
        Debug.Print("Fehler: " & ex.Message)
    Catch ex As Exception
        Debug.Print("Fehler: " & ex.Message)
    Finally
        Debug.Print("Restarbeiten")
    End Try
End Sub
```

Die Anweisungen im **Finally**-Zweig werden auf jeden Fall ausgeführt – unabhängig davon, ob eine Ausnahme ausgelöst wird oder nicht.

Ausnahmetyp ermitteln

Wenn Sie programmieren, werden Sie vermutlich zunächst mit dem allgemeinen Ausnahmetyp **Exception** arbeiten. Um gezielt Ausnahmen zu behandeln, müssen Sie dann wissen, welche Ausnahme genau ausgelöst wird.

Wir wir oben schon angedeutet haben, gibt es eine Hierarchie der Ausnahme-Objekte. Das oberste Objekt ist **Exception**, darunter folgen dann weitere wie zum Beispiel **DivideByZeroException**. Es gibt auch mehrere Ebenen – zum Beispiel gibt es unter der **Exception**-Ausnahme eine **IOException**-Ausnahme und darunter wieder die **FileNotFoundException**-Ausnahme. In den **Catch**-Zweigen des **Try...End Try**-Blocks arbeiten Sie die Ausnahmen wie oben gezeigt in der umgekehrten Reihenfolge ab.

Doch zurück zum Ermitteln der spezifischen Ausnahme für einen Fehler – diesen können Sie einfach über die Eigenschaft **GetType** ermitteln:

XAML-Eigenschaften beim Debuggen testen

Visual Studio 2019 und höher

XAML bietet sehr viele Möglichkeiten, das Design von Steuerelementen in einer Anwendung anzupassen. Viele der Eigenschaften sind von den Standardsteuerelementen vorgegeben, andere passen Sie selbst durch Definition des XAML-Codes an. Wenn dann beim Testen der Anwendung Elemente nicht wie erwartet aussehen, stellt sich die Frage, woran das liegt. Dann lautet die Aufgabe, herauszufinden, woher die jeweilige Einstellung stammt und diese so anzupassen, dass das Design wie gewünscht erscheint. Für solche Zwecke gibt es in Visual Studio eingebaute Tools, die wir uns in diesem Artikel ansehen. Mit Visual Studio 2019 gibt es viele Verbesserungen, die ein Update lohnenswert machen.

Logical Tree vs. Visual Tree

Es gibt zwei verschiedene Hierarchien bei der Darstellung von XAML-Benutzeroberflächen. Die erste kennen Sie sehr gut – dabei handelt es sich um den Logical Tree. Der Logical Tree besteht aus den Elementen, die Sie im XAML-Editor definieren – also Elemente wie **Window**, **Grid**, **StackPanel**, **TextBox**, **Button** und so weiter.

Der Logical Tree definiert dabei nicht nur, welche Elemente in der XAML-Definition enthalten sind, sondern auch deren hierarchische Anordnung.

Die zweite Hierarchie ist der Visual Tree. Warum benötigen wir eine zweite Hierarchie? Weil die Elemente des Logical Tree auf Vorlagen basieren, die nicht nur aus einem einzigen Element bestehen, sondern aus mehreren. Der Logical Tree ist also eine vereinfachte Form des Visual Tree. Wenn der Logical Tree gerendert wird, also wenn die Darstellung der Benutzeroberfläche daraus generiert wird, entstehen daraus meist sehr viel mehr Elemente. Ein **ListView**-Steuerelement etwa besteht aus Rändern, Bildlaufleisten, Spaltenköpfen, Zeilen, Spalten und weiteren Elementen.

Warum müssen wir das an dieser Stelle wissen? Weil wir zur Laufzeit nicht auf die Eigenschaften der Elemente des Logical Trees zugreifen können, aber auf die Eigenschaften des Visual Trees. Wie das gelingt, zeigen wir in den folgenden Abschnitten.

Debugging-Funktionen aktivieren

Damit wir die nachfolgend vorgestellten Funktionen nutzen können, müssen wir diese zunächst in den Optionen von Visual Studio aktivieren. Dazu öffnen Sie den Optionen-Dialog mit dem Menübefehl **Extras|Optionen**. Im Optionen-Dialog wechseln Sie dann zum Bereich **Debugging|Allgemein**. Hier finden Sie die Option **UI-Debugtools für XAML aktivieren** (siehe Bild 1).

Die einzelnen Einstellungen haben die folgende Bedeutung:

- **UI-Debugtools für XAML aktivieren**: Mit dieser Option legen Sie fest, ob die Debugtools überhaupt aktiviert sein sollen. Die untergeordneten Optionen steuern dann, wie die Debug-Tools bereitgestellt werden.

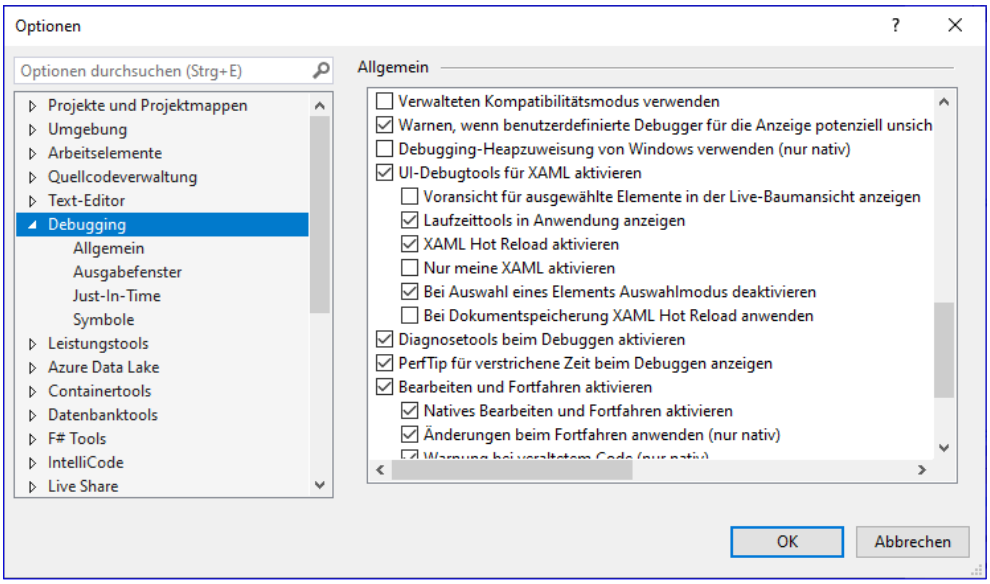


Bild 1: Einstellungen für das Debuggen von XAML-Code

- **Voransicht für ausgewählte Elemente in der Live-Baumansicht anzeigen**: Das aktuell ausgewählte XAML-Element wird im Bereich **Visueller Livebaum** angezeigt.

- **Laufzeittools in Anwendung anzeigen**: Blendet eine Menüleiste oben im Fenster der zu testenden Anwendung ein, über welche die Debugging-Funktionen schnell aufgerufen werden können (siehe Bild 2). Die Laufzeittools können Sie übrigens auch zur Laufzeit durch Aktivieren der Option einblenden.

- **XAML Hot Reload aktivieren**: Aktiviert die Funktion XAML Hot Reload, die dazu dient, zur Laufzeit Änderungen am XAML-Code vorzunehmen und diese direkt in Echtzeit zu testen.

- **Nur meine XAML aktivieren**: Der visuelle Livebaum zeigt ab Visual Studio 2019 (Version 16.4) nur benutzerdefinierte Elemente an, wenn diese Option aktiviert ist. Wenn Sie alle Elemente sehen wollen, deaktivieren Sie diese Option.

- **Bei Auswahl eines Elements Auswahlmodus deaktivieren**: Wenn Sie bei aktivierter Option in der Menüleiste für die Debugging-Tools in der Anwendung auf die Schaltfläche **Element auswählen** klicken und dann über die Elemente der Anwendung fahren, wird das jeweils überfahrene Element mit einem gestrichelten, roten Rahmen versehen. Klicken Sie es an,

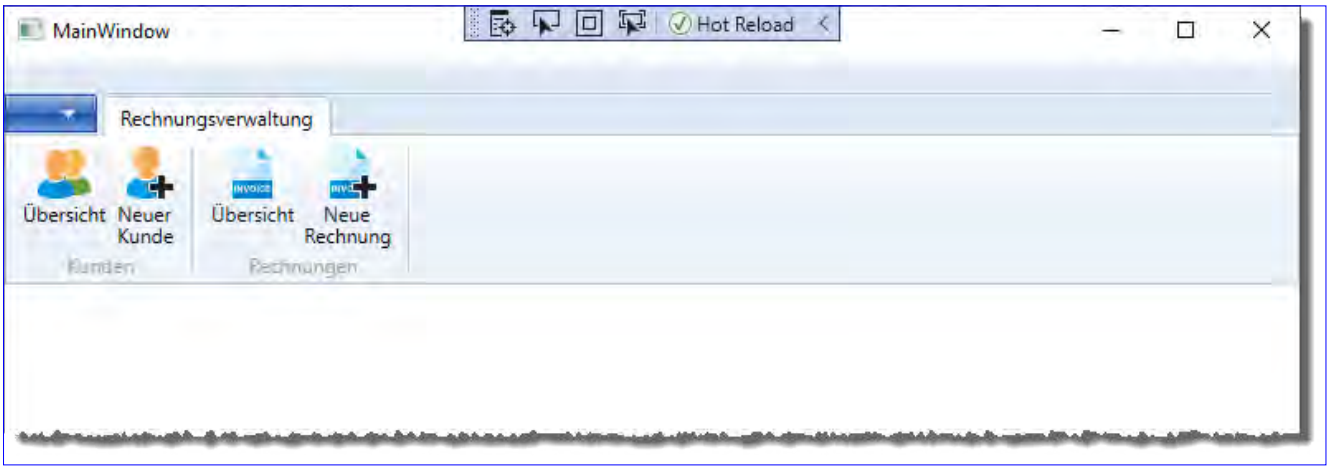


Bild 2: Steuerelemente für das Debuggen von XAML-Eigenschaften

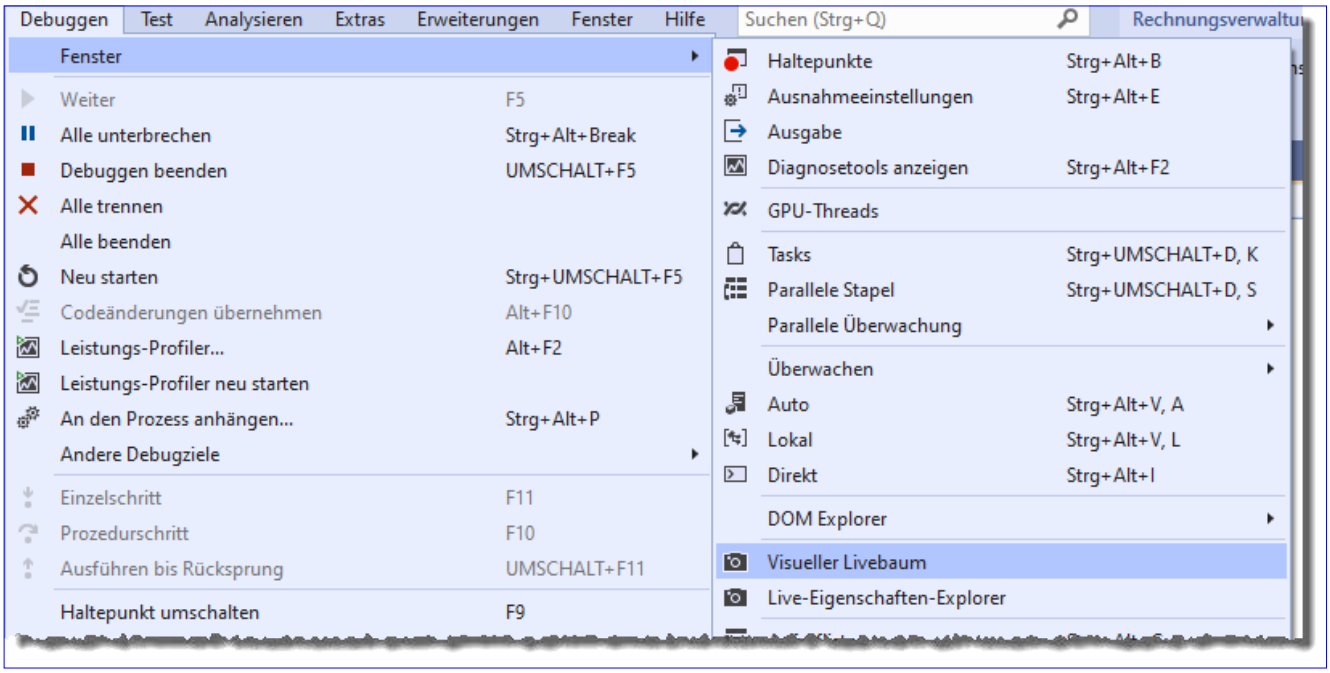


Bild 3: Visuellen Livebaum und Live-Eigenschaften-Explorer per Menü aktivieren

wird das jeweilige Element im visuellen Livebaum hervorgehoben. Danach wird die Schaltfläche Element auswählen wieder deaktiviert und Sie müssen diese für die erneute Auswahl eines Elements erneut anklicken. Ist die Option deaktiviert, können Sie nach dem Anklicken der Schaltfläche **Element auswählen** immer wieder neue Elemente im visuellen Livebaum anzeigen lassen. Die Schaltfläche **Element auswählen** bleibt dann aktiviert.

- **Bei Dokumentspeicherung XML Hot Reload anwenden**: XML Hot Reload wird angewendet, wenn Sie das Dokument speichern.

Visuellen Livebaum und Eigenschaften-Explorer aktivieren

Wenn Sie die Symbole zum Aufrufen der Debugging-Funktionen im Anwendungsfenster nicht sehen wollen, können Sie die wichtigsten Funktionen auch über das Menü von Visual Studio aufrufen. Dort finden Sie die Einträge **Debuggen|Fenster|Visueller Livebaum** und **Debuggen|Fenster|Live-Eigenschaften-Explorer** (siehe Bild 3).

Den visuellen Livebaum nutzen

Den visuellen Livebaum haben wir nun schon mehrfach erwähnt, aber wir haben ihn uns noch nicht angesehen. Den logischen Baum kennen wir, also schauen wir uns nun den visuellen Livebaum an. Diesen aktivieren Sie beispielsweise über den oben genannten Menübfehl **Debuggen|Fenster|Visueller Livebaum** oder, wenn das Laufzeittools-Menü im Anwendungsfenster aktiviert ist, über die Schaltfläche **Zur visuellen Echtzeitstruktur wechseln** (linke Schaltfläche). Wir haben ein einfaches Fenster mit einer Schaltfläche erstellt, deren Code im logischen Baum vereinfacht wie folgt aussieht:

```
<Window x:Class="MainWindow" ... Title="MainWindow" Height="450" Width="800">
    <Window.Resources>
        <Style TargetType="Button">
```

```
            <Setter Property="Margin" Value="5"></Setter>
        </Style>
    </Window.Resources>
    <Grid>
        <Grid.ColumnDefinitions>
            <ColumnDefinition Width="Auto"></ColumnDefinition>
            <ColumnDefinition Width="*"></ColumnDefinition>
        </Grid.ColumnDefinitions>
        <Grid.RowDefinitions>
            <RowDefinition Height="Auto"></RowDefinition>
            <RowDefinition Height="*"></RowDefinition>
        </Grid.RowDefinitions>
        <Button x:Name="btnTest">Beispielschaltfläche</Button>
    </Grid>
</Window>
```

Aktivieren wir nun den visuellen Livebaum mit einer der beiden genannten Methoden, sieht dieser zunächst wie in Bild 4 aus.

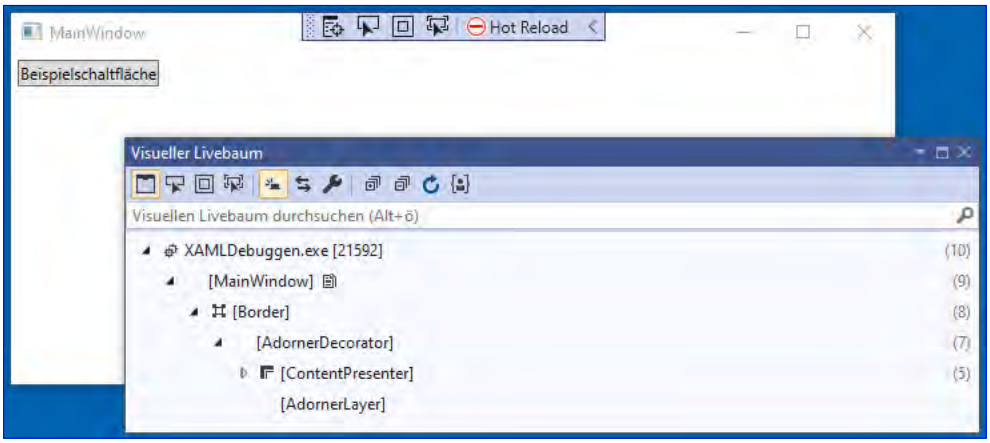


Bild 4: Visueller Livebaum für ein Fenster mit einer Schaltfläche

Nun wollen wir ein Element auswählen und klicken dazu zunächst die Schaltfläche **Element auswählen** an. Fahren wir nun über die Schaltfläche, wird diese mit einem roten, gestrichelten Rand versehen (siehe Bild 5).

Hier ist ein wenig Fingerspitzengefühl gefragt, denn zum Markieren der Schaltfläche müssen Sie genau den Bereich zwischen dem Rand und der Beschriftung der Schaltfläche erwischen. Und das ist bereits ein Beispiel für den Unterschied zwischen den Elementen

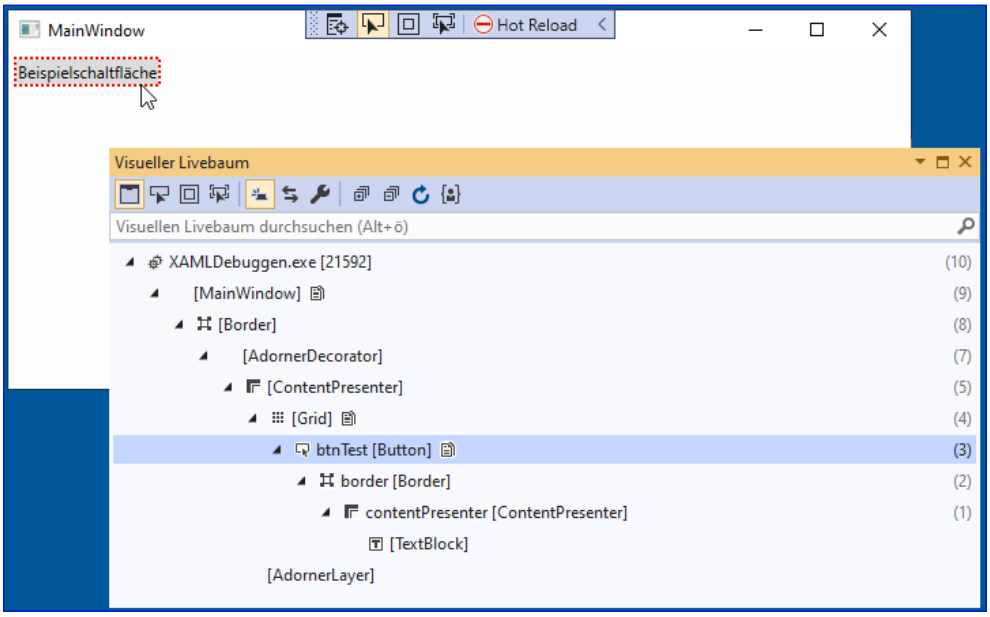


Bild 5: Visueller Livebaum nach dem Überfahren der Schaltfläche

FlowDocumente mit VB zusammenstellen

FlowDocument-Element werden per RichTextBox-Feld eingegeben, per XAML definiert oder auch per VB zusammengestellt. Letzteres ist vor allem dann interessant, wenn Sie das Dokument mit den Daten aus einer externen Quelle wie einer Datenbank füllen wollen. So können Sie beispielsweise Rechnungen erstellen, Auflistungen von Produkten, Angebote und beliebige andere Dokumente. Diese können Sie dann als PDF verschicken oder ausdrucken. Dieser Artikel zeigt die Grundlagen zum Erstellen von FlowDocument-Objekten und zum Füllen mit den gewünschten Elementen und Inhalten.

In anderen Artikeln wie **FlowDocument-Elemente mit XAML** (**www.datenbankentwickler.net/203**) oder **Das RichTextBox-Steuerelement** (**www.datenbankentwickler.net/204**) haben wir die Grundlagen zu **FlowDocument**-Elementen geliefert und zu den Steuerelementen, in denen sie angezeigt und bearbeitet werden können.

Nun wollen wir zeigen, wie Sie Elemente, die Sie in den zuvor genannten Beiträgen mit XAML-Definitionen erstellt oder über ein **RichTextBox**-Steuerelement eingegeben haben, per VB-Code erstellen können.

Vorbereitung

Die Basis unseres Beispiels definieren wir wie folgt. Wir haben ein Grid, in dessen erster Zeile ein **StackPanel**-Element liegt, das wir im Laufe des Artikels mit Schaltflächen zum Ausführen unserer Beispiele füllen werden. In der zweiten **Grid**-Zeile haben wir ein **FlowDocumentPageViewer**-Element zur Anzeige der erzeugten **FlowDocument**-Elemente definiert:

```
<Grid>
    <Grid.RowDefinitions>
        <RowDefinition Height="Auto"></RowDefinition>
        <RowDefinition Height="*"></RowDefinition>
    </Grid.RowDefinitions>
    <StackPanel Orientation="Horizontal">
        <Button x:Name="btnEinfacherText" Click="BtnEinfacherText_Click">Einfacher Text</Button>
    </StackPanel>
    <FlowDocumentPageViewer x:Name="fdp" Margin="3" Grid.Row="1"></FlowDocumentPageViewer>
</Grid>
```

Die Beispielmethoden, die durch die Schaltflächen ausgelöst werden, legen wir im Code behind-Modul des Fensters an. Für den ersten Beispieltext hinterlegen wir die folgende Prozedur in der Klasse **FlowDocumentPerVB**:

```
Public Class FlowDocumentPerVB
    Private Sub BtnEinfacherText_Click(sender As Object, e As RoutedEventArgs)
        Dim objFlowDocument As FlowDocument
```

```
        Dim objParagraph As Paragraph
        objFlowDocument = New FlowDocument
        objParagraph = New Paragraph()
        With objParagraph
            .Inlines.Add("Dies ist ein erster Absatz.")
        End With
        objFlowDocument.Blocks.Add(objParagraph)
        fdp.Document = objFlowDocument
    End Sub
End Class
```

Klicken wir nach dem Starten des Projekts auf die Schaltfläche, erhalten wir das Ergebnis aus Bild 1. Hier haben wir noch keine Absatzformate definiert. Die Zeilen, die dem Paragraph-Objekt den Text hinzufügen, können wir auch noch abkürzen durch folgenden Ausdruck:

```
objParagraph = New Paragraph(New Run("Dies ist ein erster Absatz."))
```

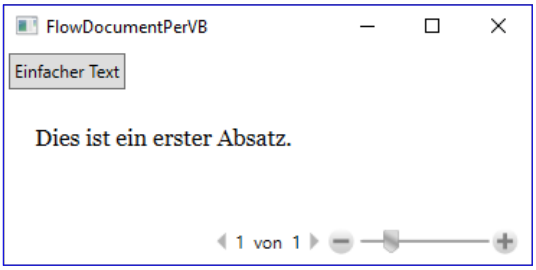


Bild 1: Erster Beispieltext

Absatz mit Formatierungen

Im zweiten Beispiel wollen wir einen Absatz mit Formatierungen versehen – der Text soll in der Schriftart **Calibri** in der Größe **24** mit fetter Auszeichnung ausgegeben werden.

Dazu verwenden wir die folgende Methode, die durch die Schaltfläche mit der Beschriftung **Text mit Format** ausgelöst wird:

```
Private Sub BtnTextMitFormat_Click(sender As Object, e As RoutedEventArgs)
    Dim objFlowDocument As FlowDocument
    Dim objParagraph As Paragraph
    objFlowDocument = New FlowDocument
    objParagraph = New Paragraph(New Run("Dies ist ein erster Absatz."))
    With objParagraph
        .FontFamily = New FontFamily("Calibri")
        .FontSize = 24
        .FontWeight = FontWeights.Bold
    End With
    objFlowDocument.Blocks.Add(objParagraph)
    fdp.Document = objFlowDocument
End Sub
```

Bild 2: Text mit Formatierungen

Das Ergebnis sehen Sie in Bild 2. Die Formatierungen können wir einfach über die Eigenschaften des **Paragraph**-Objekts zuweisen. Gegenüber Programmiersprachen wie VBA lässt sich hier jedoch nicht in jedem

Fall einfach der Name der Schriftart zuweisen, sondern wir müssen dazu neue Objekte erstellen wie **FontFamily** oder Auflistungen aus Klassen verwenden wie **FontWeights.Bold**.

Tabelle mit Elementen erstellen

Das Erstellen einer einfachen Tabelle ist schon etwas aufwendiger. Schauen wir uns ein einfaches Beispiel an.

Dazu wollen wir die Tabelle aus Bild 3 erstellen – also eine Tabelle mit drei Spalten und vier Zeilen sowie Inhalt und Rahmenlinien.

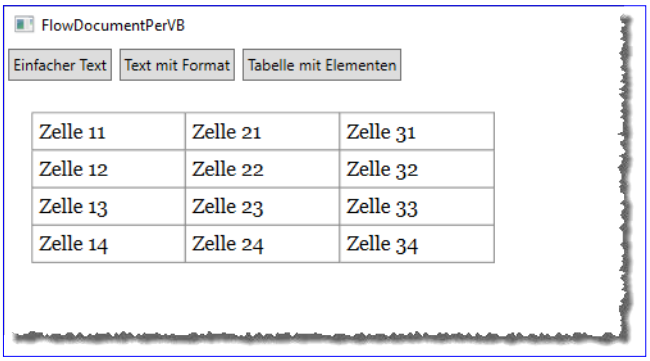


Bild 3: Erstellen einer einfachen Tabelle

Wir deklarieren zunächst einige Variablen, die wir für die Erstellung der Tabelle benötigen sowie die zwei Laufvariablen **i** und **j**, mit denen wir die Zeilen und Spalten zum Erstellen der Zellen durchlaufen wollen:

```
Private Sub BtnTabelle_Click(sender As Object, e As RoutedEventArgs)
    Dim objFlowDocument As FlowDocument
    Dim objTable As Table
    Dim objTableColumn As TableColumn
    Dim objTableRowGroup As TableRowGroup
    Dim objTableRow As TableRow
    Dim objTableCell As TableCell
    Dim objParagraph As Paragraph
    Dim i As Integer
    Dim j As Integer
```

Dann erstellen wir ein neues **FlowDocument**- und ein neues **Table**-Element. Das **Table**-Element fügen wir über die **Add**-Methode der **Blocks**-Auflistung zum **FlowDocument** hinzu:

```
    objFlowDocument = New FlowDocument
    objTable = New Table
    objFlowDocument.Blocks.Add(objTable)
```

Für das **Table**-Objekt können wir nun bereits allgemeine Einstellungen vornehmen, die sich auch auf die darunter liegenden Elemente vererben können wie hier die Schriftart und die Schriftgröße. Eine weitere Einstellung namens **CellSpacing** gibt den Abstand zwischen den Zellen an. Diesen müssen wir auf **0** einstellen, wenn wir durchgehende Rahmenlinien für die einzelnen Zeilen und Spalten erhalten wollen:

```
    With objTable
        .FontFamily = New FontFamily("Calibri")
        .FontSize = 12
```

```
        .CellSpacing = 0
    End With
```

Bevor wir Zellen anlegen können, müssen wir der Tabelle Spalten und Zeilen hinzufügen. Bei den Spalten gelingt das etwas einfacher, denn hier brauchen wir nur neue **TableColumn**-Elemente zu erstellen und diese dann über die **Add**-Methode der **Columns**-Auflistung zur Tabelle aus **objTable** hinzuzufügen. Hier fügen wir in einer **For...Next**-Schleife drei Spalten zur Tabelle hinzu:

```
    For i = 1 To 3
        objTableColumn = New TableColumn
        objTable.Columns.Add(objTableColumn)
    Next
```

Um Zeilen hinzuzufügen, benötigen wir zunächst ein **TableRow**-Objekt, das wir erstellen und der **RowGroups**-Auflistung hinzufügen:

```
    objTableRowGroup = New TableRowGroup
    objTable.RowGroups.Add(objTableRowGroup)
```

Die Tabellenzellen erstellen wir in einem Zuge mit den Tabellenzeilen. Dazu durchlaufen wir eine **For...Next**-Schleife mit der Variablen **j** für die Zeilen. Hier erstellen wir ein neues **TableRow**-Objekt und weisen es der **Rows**-Auflistung der **TableRowGroup** zu:

```
    For j = 1 To 4
        objTableRow = New TableRow
        objTableRowGroup.Rows.Add(objTableRow)
```

Damit können wir in dieser Zeile die Tabellenzellen anlegen und mit Inhalt füllen. Dabei nehmen wir direkt ein paar Formatierungen vor. Wir legen mit **Padding** den Abstand zum enthaltenen Text fest. Mit **BorderBrush** definieren wir die Rahmenfarbe. Danach folgen einige Bedingungen, mit denen wir ermitteln, in welcher Zeile und Spalte wir uns gerade befinden.

In der obersten Zeile soll der Rahmen oben und unten und in der ersten Spalte links und rechts gesetzt werden, in den folgenden Zeilen soll der Rahmen nur unten und in den folgenden Spalten nur rechts gesetzt werden. Anderenfalls erscheint der Rahmen dicker:

```
        For i = 1 To 3
            objTableCell = New TableCell()
            objTableCell.Padding = New Thickness(5)
            objTableCell.BorderBrush = Brushes.Gray
            If j = 1 Then
                If i = 1 Then
```

XML mit .NET

XML ist eines der bekanntesten Datenaustauschformate. XML bedeutet Extensible Markup Language und erlaubt das Speichern von Daten in Textdateien in einem hierarchisch strukturierten Format. Dies wird zum Beispiel in den Konfigurationsdateien eines Visual Studio-Projekts genutzt – und auch XAML, die Sprache zum Definieren von WPF-Benutzeroberflächen, ist auf XML aufgebaut. Wenn Daten zwischen zwei nicht kompatiblen Systemen ausgetauscht werden sollen, kommt oft XML zum Einsatz. Auch wenn Sie Daten von einem Webservice abrufen wollen, kommen die Daten in vielen Fällen im XML-Format. Je nach Webservice können Sie diesem auch Daten im XML-Format schicken, die dann etwa durch Speichern in einer Datenbank weiterverarbeitet werden. Grund genug, sich XML einmal genauer anzusehen und zu schauen, welche Möglichkeiten .NET für das Lesen, Bearbeiten und Erstellen von XML-Dokumenten bietet.

Grundlagen zu XML

XML-Dokumente enthalten strukturierte Daten im ASCII-Format – so können Anwendungen von verschiedenen Plattformen den Inhalt verstehen. Dementsprechend können Sie XML-Dokumente auch in einem einfachen Texteditor öffnen und bearbeiten. XML-Dokumente, die als Datei gespeichert werden, erhalten die Dateiendung **.xml**, also beispielsweise **Kunden.xml**.

Ein XML-Dokument in Visual Studio anlegen

Auch wenn XML-Dokumente keinen speziellen Editor benötigen, wollen wir die Fähigkeiten von Visual Studio nutzen. Also öffnen Sie Visual Studio und legen mit **Datei|Neu|Datei...** eine neue Datei an. Wählen Sie im nun erscheinenden Dialog **Neue Datei** den Eintag **XML-Datei** aus. Dies fügt eine neue, leere XML-Datei namens **XMLFile1.xml** hinzu, die im Verzeichnis **C:\Users\<Benutzername>\source\repos** gespeichert wird. Erfreulicherweise werden Fehler im XML-Dokument von Visual Studio im Bereich **Fehlerliste** angezeigt (siehe Bild 1). Hier sehen Sie direkt, dass es durchaus Anforderungen an XML-Dokumente gibt – in diesem Fall wird das Fehlen eines Stammelements bemängelt.

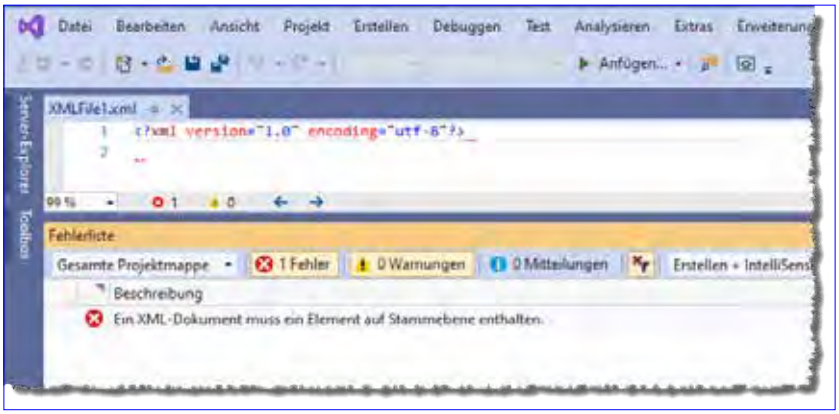


Bild 1: Ein neues, leeres XML-Dokument

XML-Dokument als solches definieren

Damit ein XML-Dokument als solches erkannt werden kann, geben wir in der ersten Zeile einen sogenannten Prolog an. Dieser hat eine andere Syntax als die übrigen XML-Zeilen, der Ihnen von der Programmiersprache PHP bekannt vorkommen könnte:

```
<?xml version="1.0" ?>
```

Hier können Sie zusätzlich noch die Kodierung des Dokuments angeben, hier zum Beispiel **utf-8**:

```
<?xml version="1.0" encoding="utf-8"?>
```

Elemente eines XML-Dokuments

Wie im Screenshot oben zu sehen, enthält ein XML-Dokument neben der Zeile mit der XML-Version und der Kodierung also mindestens ein Element auf der obersten Ebene. Ein solches Element hat entweder ein öffnendes und ein schließendes Tag-Element (Starttag und Endtag) oder ein abgeschlossenes Tag-Element (Leertag). Zwischen dem öffnenden und dem schließenden Tag-Element befinden sich untergeordnete Elemente.

Ein abgeschlossenes Tag-Element enthält keine weiteren untergeordneten Elemente. Ein Element mit öffnendem und schließendem Tag sieht wie folgt aus:

```
<Beispielelement></Beispielelement>
```

Der Name des Elements muss im öffnenden wie im schließenden Tag genau gleich geschrieben sein – das gilt auch für Groß- und Kleinschreibung. Das geschlossene Element sieht so so:

```
<Beispielelement />
```

Das Element wird durch den Schrägstrich vor dem Größer-Zeichen als abgeschlossen markiert. Beide Varianten reichen Visual Studio schon aus, um die Fehlermeldung verschwinden zu lassen. Jedes offene Element kann ein oder mehrere Unterelemente enthalten:

```
<Hauptelement>
    <Unterelement />
    <Unterelement></Unterelement>
</Hauptelement>
```

Hauptelement und **Unterelement** sind die Namen der Elemente. Die eigentlichen Daten werden an zwei Stellen gespeichert: in Attributen oder als Werte von Elementen. Attribute werden innerhalb des öffnenden Tags hinter dem Elementnamen angegeben, und zwar durch Attributname und Attributwert. Jeder Attributname darf nur einmal je Element vorkommen, der Attributwert muss in Hochkommata oder Anführungszeichen angegeben werden:

```
<Hauptelement Beispielattribut="Beispielwert"></Hauptelement>
```

Bei geschlossenen Elementen landet das Attribut ebenfalls hinter dem Elementnamen:

```
<Hauptelement Beispielattribut="Beispielwert" />
```

Außerdem können Sie für jedes Element alternativ zu weiteren untergeordneten Elementen auch eine Zeichenkette als Wert des Elements angeben:

```
<Elementname>Elementinhalt</Elementname>
```

Ein Element mit einer Zeichenkette als Inhalt kann keine weiteren Elemente enthalten.

Elementnamen verwenden

Für die Elementnamen gelten bestimmte Regeln:

- Der Elementname folgt unmittelbar auf die öffnende spitze Klammer beziehungsweise beim schließenden Tag hinter dem Schrägstrich.
- Der Elementname darf keine Leerzeichen enthalten.
- Der Elementname darf Buchstaben, Ziffern, Unterstrich, Bindestrich, Punkt und Doppelpunkt enthalten. Doppelpunkte dürfen allerdings nur hinter Namespaces angegeben werden – mehr dazu weiter unten. Als erstes Zeichen dürfen nur Buchstaben zum Einsatz kommen.
- Der Elementname darf nicht mit einer Ziffer oder mit der Zeichenfolge **xml** beginnen (letzteres wird von Visual Studion nicht bemängelt).

Wohlgeformtheit von XML-Dokumenten

Die wichtigsten Kriterien für die Wohlgeformtheit eines XML-Dokuments lauten also:

- Das Dokument muss eine Verarbeitungsanweisung enthalten (**<?xml version="1.0"?>**).
- Das Dokument besitzt in der ersten Ebene genau ein Element.
- Jedes Element muss geschlossen werden (**<Beispielelement></Beispielelement>** oder **<Beispielelement />**)
- Start- und Endtags müssen den gleichen Namen haben, auch bezüglich der Groß- und Kleinschreibung.
- Die Start- und Endtags werden paarweise verschachtelt, das heißt, alle innerhalb eines übergeordneten Elements angelegten Unterelemente müssen erst wieder geschlossen werden, bevor das übergeordnete Element geschlossen wird.

Unterschied zwischen Attributen und Elementen

Wenn Sie als Wert eine einfache Zeichenkette verwenden wollen, können Sie diese gleichwertig für ein Attribut oder für ein Unterelement angeben. Ein **Kunde**-Element mit **Vorname** und **Nachname** würde mit Attributen so aussehen:

```
<Kunde Vorname="Andre" Nachname="Minhorst"></Kunde>
```

Sie können **Vorname** und **Nachname** jedoch auch in untergeordneten Elementen darstellen:

```
<Kunde>
  <Vorname>Andre</Vorname>
  <Nachname>Minhorst</Nachname>
</Kunde>
```

Wenn ein Element allerdings einen Text enthalten soll, können Sie diesem keine weiteren Unterelemente hinzufügen – dann müssen Sie zusätzliche Informationen als Attribut angeben:

```
<Artikel Artikelnummer="123">Access im Unternehmen</Artikel>
```

Kommentare

Wenn Sie Kommentare zu einem XML-Dokument hinzufügen wollen, beginnen Sie diesen mit **<!--** und beenden ihn mit **-->**, also genau wie unter HTML:

```
<!--Dies ist ein Kommentar.-->
```

Texte in Elementen

Wenn ein Element oder ein Attribut eines Elements einen Text enthält, können Sie dafür alle möglichen Zeichen verwenden – mit wenigen Ausnahmen, nämlich dem Größer- und dem Kleiner-Zeichen, dem Und-Zeichen, dem Hochkomma und dem Anführungszeichen. Diese müssen Sie durch bestimmte andere Zeichenkombinationen ersetzen:

- Und-Zeichen (**&**): Ersetzen durch **&**
- Kleiner-Zeichen (**<**): Ersetzen durch **<**
- Größer-Zeichen (**>**): Ersetzen durch **>**
- Hochkomma (**'**): Ersetzen durch **'**
- Anführungszeichen (**"**): Ersetzen durch **"**

Sonderzeichen trotzdem verwenden mit CData

Wenn Sie Sonderzeichen wie die oben beschriebenen dennoch verwenden wollen, gibt es eine Alternative. Dazu verwenden Sie einen **CData**-Abschnitt als Inhalt eines Elements. Der **CData**-Abschnitt beginnt mit **<![CDATA[** und endet mit **]]>**.

Dazwischen können Sie Texte mit beliebigen Zeichen platzieren, also auch solche wie die oben genannten Sonderzeichen:

```
<ElementMitSonderzeichen><![CDATA[Die Sonderzeichen <, >, &, ', " sind hier erlaubt.]]></ElementMitSonderzeichen>
```

Die einzige Ausnahme ist die Zeichenfolge **]]>**, denn diese zeigt das Ende des **CData**-Abschnitts an. Der Rest des eigentlichen **CData**-Abschnitts würde dann als Fehler interpretiert werden. Die Verwendung von **CData**-Abschnitten eignet sich beispielsweise, um ein XML-Dokument innerhalb des Elements eines XML-Dokuments einzufügen.

Namespaces für eindeutige Zuordnung verwenden

Namespaces verwendet man, um Verwechslungen gleichnamiger Elemente zu verhindern. Wenn Sie beispielsweise in einem XML-Dokument sowohl Kunden als auch Lieferanten verwalten und beide enthalten Elemente namens Firma, dann könnte es beim Zugriff auf dieses Element zu Verwechslungen kommen. Es kann auch sein, dass Sie in Ihren eigenen XML-Dokumenten sehr genau darauf achten, dass alle Elemente eine eindeutige Bezeichnung haben. Vielleicht fügt aber ein Nutzer Ihres XML-Dokuments dieses mit einem anderen XML-Dokument zusammen und erhält dann doppelte Bezeichnungen für Elemente mit unterschiedlichen Bedeutungen. In folgendem Beispiel wird das deutlich:

```
<?xml version="1.0" encoding="utf-8"?>
<Bestellverwaltung>
  <Kunden>
    <Kunde>
      <Firma>André Minhorst Verlag</Firma>
    </Kunde>
    ...
  </Kunden>
  <Lieferanten>
    <Lieferant>
      <Firma>Beispiellieferant GmbH</Firma>
    </Lieferant>
    ...
  </Lieferanten>
</Bestellverwaltung>
```

Suchen Sie hier nun nach allen Elementen mit dem Namen **Firma**, erhalten Sie sowohl die Firmenangaben für die Kunden als auch für die Lieferanten. Um das zu ändern, definieren wir Namespaces. Das können wir direkt im Stammelement **Bestellverwaltung** erledigen. Wenn wir das tun, können die Namespaces im gesamten XML-Dokument verwendet werden. Die Namespaces werden in Form von Attributen definiert, für die bestimmte Regeln gelten. Die beiden Namespaces für die Kunden und die Lieferanten würden wir wie folgt definieren:

```
<Bestellverwaltung xmlns:k="http://minhorst.com/namespaces/kunde"
                   xmlns:l="http://minhorst.com/namespaces/lieferant">
```

Der Name des Attributs zum Festlegen eines Namespaces beginnt immer mit **xmlns**. Wenn Sie einen Namespace definieren wollen, der für alle Elemente unter dem Element mit der Namespace-Definition gilt, geben Sie als Attributname nur **xmlns** an. Mit der folgenden Angabe fügen Sie alle Elemente des XML-Dokuments, die nicht anderes zugeordnet sind, dem Standardnamespace **http://minhorst.com/namespaces/bestellverwaltung** zu:

XML-Dokumente schnell lesen mit XmlReader

Unter .NET gibt es verschiedene Methoden und Klassen, mit denen Sie auf die Daten in XML-Dokumenten zugreifen können. Diese nutzen Sie je nach Einsatzzweck. Wenn Sie sehr große Dateien einlesen wollen, bietet sich die XmlReader-Klasse an. Mit dieser durchlaufen Sie das XML-Dokument sequenziell, das heißt, Element für Element. Dieser Artikel zeigt, wie Sie mit den Methoden der XmlReader-Klasse auf die Elemente eines XML-Dokuments zugreifen. Außerdem schauen wir uns an, wie Sie die eingelesen Inhalte in Objekte schreiben können, über die Sie die Inhalte dann in die Tabellen einer Datenbank eintragen.

Wer unter Access/VBA auf XML-Dokumente zugegriffen hat, hat dazu in der Regel das Document Object Model verwendet und dabei die Elemente mit den gewünschten Daten gezielt angesteuert. Unter VBA gibt es noch eine weitere Technik namens SAX. Diese ist recht kompliziert und wird sehr selten verwendet. Dieser Technik kommt der hier verwendete **XmlReader** jedoch recht nahe. Seine Handhabung ist allerdings einfacher als bei SAX.

Dokument öffnen

Um ein XML-Dokument zu öffnen, benötigen wir ein Objekt auf Basis der **XMLReader**-Klasse. Diese ist Teil des Namespaces **System.XML**, den wir mit der folgenden Anweisung etwa in der Code behind-Klasse eines Fensters einer neuen Desktop-Anwendung auf Basis von XAML und VB referenzieren:

```
Imports System.Xml
```

Danach können wir auf die **XmlReader**-Klasse und ihre Elemente zugreifen. Bei der Klasse handelt es sich um eine abstrakte Klasse, das heißt, Sie können diese nicht instanzieren. Wir können aber dennoch eine Objektvariable nutzen, um auf das zu lesende XML-Dokument zuzugreifen. Um ein XML-Dokument zu öffnen, das auf der Festplatte liegt, verwenden Sie die **Create**-Methode. Dieser übergeben wir den Pfad zu der einzulesenden XML-Datei:

```
Dim objReader As XmlReader
Dim strPfad As String
strPfad = "c:\...\Kunden.xml"
objReader = XmlReader.Create(strPfad)
```

Danach befindet sich das XML-Dokument im Zugriff. Wir schauen uns die Funktionsweise der Klasse **XmlReader** anhand des folgenden XML-Dokuments an:

```
<?xml version="1.0" encoding="utf-8"?>
<Bestellverwaltung>
  <!--Auflistung der Kunden-->
```

```
  <Kunden>
    <Kunde ID="1">
      <Firma>André Minhorst Verlag</Firma>
      <Vorname>André</Vorname>
      <Nachname>Minhorst</Nachname>
      <Beschreibung><![CDATA[Beschreibung mit CDATA]]></Beschreibung>
    </Kunde>
    <Kunde ID="2">
      <Firma>Müller GmbH</Firma>
      <Vorname>Klaus</Vorname>
      <Nachname>Müller</Nachname>
    </Kunde>
  </Kunden>
</Bestellverwaltung>
```

XML-Dokument sequenziell durchlaufen mit Read

Im Gegensatz zum Durchlaufen einer Textdatei durchlaufen wir hier nicht Zeile für Zeile, sondern Element für Element. Ein Element ist dabei nicht etwa eine Einheit wie **<Vorname>André</Nachname>**. In diesem Fall sind **<Vorname>**, **André** und **</Vorname>** jeweils ein Element. Auch der Prolog des XML-Dokuments ist ein eigenes Element. Auf diese Elemente greifen wir mit der **Read**-Methode zu. Nachdem wir diese erstmals aufgerufen haben, referenziert **objReader** das erste Element, in diesem Fall den Prolog. Danach geht es weiter mit den folgenden Elementen – also den Starttags, den enthaltenen Texten, den Endtags und auch den leeren Elementen dazwischen, den sogenannten **Whitespace**-Elementen. Wie erkennen wir nun, welchen Elementtyp **objReader** gerade geladen hat? Dazu können wir die Eigenschaft NodeType verwenden. Diesen lassen wir uns in einer ersten Schleife ausgeben:

```
Do While objReader.Read
    Debug.Print(objReader.NodeType)
Loop
```

Wir durchlaufen also alle Elemente in einer **Do While**-Schleife, deren Abbruch **objReader.Read** lautet. Die **Read**-Methode liefert den Wert **True** zurück, falls noch ein neues Element eingelesen werden konnte und den Wert **False**, wenn das Ende des Dokuments erreicht wurde. Das Ergebnis der Ausgabe ist wenig aussagekräftig, denn **NodeType** liefert einen Zahlenwert (siehe Bild 1).

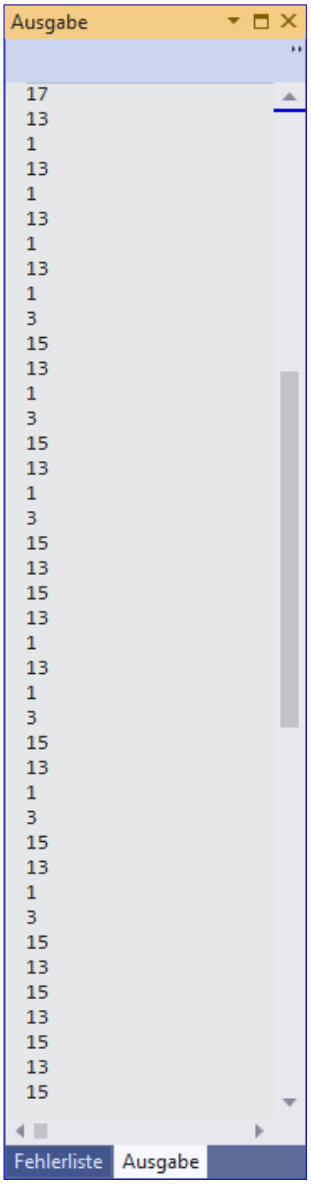


Bild 1: Elementtypen

Welcher Wert entspricht nun welchem Elementtyp? Das können wir uns im Objektkatalog ansehen, wo wir nach der Eigenschaft **NodeType** suchen und den Eintrag **System.Xml.XmlNodeType** vorfinden. Beim Anklicken der Elemente zeigt der Dialog auch die Zahlenwerte an, die wir mit denen unserer Ausgabe abgleichen können (siehe Bild 2).

Also reichern wir unsere **Do While**-Schleife um eine **Select Case**-Bedingung an, welche den Typ ermittelt und diesen im Direktbereich ausgibt. Für uns sind die folgenden Elementtypen interessant:

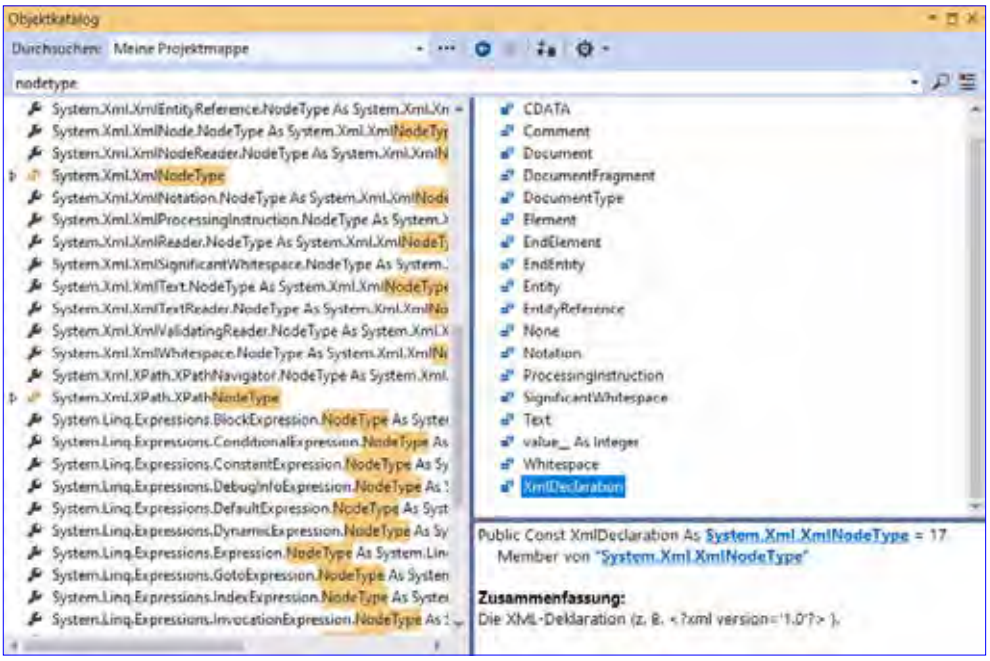


Bild 2: Elementtypen im Objektkatalog

- **Attribute**: Attribut
- **CDATA**: Block mit einem CDATA-Abschnitt
- **Comment**: Kommentar, zum Beispiel **<!--Kommentar-->**
- **Element**: Starttag, zum Beispiel **<Kunde>**
- **EndElement**: Endtag, zum Beispiel **</Kunde>**
- **Text**: Text in einem Element
- **Whitespace**: Leerraum zwischen Elementen
- **XmlDeclaration**: XML-Deklarationszeile, zum Beispiel **<?xml version='1.0'>**

In der folgenden Schleife geben wir die Typen der Elemente unseres Dokuments aus:

```
Do While objReader.Read
    Select Case objReader.NodeType
        Case XmlNodeType.XmlDeclaration
            Debug.Print("XmlDeclaration")
        Case XmlNodeType.Element
            Debug.Print("Element")
        Case XmlNodeType.EndElement
            Debug.Print("EndElement")
        Case XmlNodeType.Whitespace
            Debug.Print("Whitespace")
```

```
        Case XmlNodeType.Text
            Debug.Print("Text")
        Case Else
            Debug.Print(objReader.NodeType)
    End Select
Loop
```

Nun wollen wir für die unterschiedlichen Elementtypen entsprechende Ausgaben bewirken. Diese erhalten wir über die Eigenschaften des aktuellen Elements. **NodeType** war die erste Eigenschaft, die Sie kennengelernt haben. Es gibt jedoch noch weitere Eigenschaften, die für uns interessant sein können:

- **Depth**: Tiefe des aktuellen Elements in der Hierarchie
- **IsStartElement**: Gibt an, ob das aktuelle Element ein Starttag ist. Der folgende Code gibt beispielsweise alle Namen von Starttags aus:

```
Do While objReader.Read
    If (objReader.IsStartElement) Then
        Debug.Print(objReader.Name)
    End If
Loop
```

- **Name**: Name des Elements. Vor allem interessant bei den Elementtypen **Element** und **EndElement**.
- **NodeType**: Typ des Elements
- **Value**: Wert des Elements, vor allem interessant bei den Elementtypen **Text** oder **Comment**.
- **IsEmptyElement**: Gibt an, ob das Element einen Wert enthält.

Attribute erkennen mit HasAttributes und AttributeCount

Möglicherweise ist Ihnen aufgefallen, das wir zwar die Inhalte der Elemente ausgegeben haben, jedoch noch nicht die der Attribute. Diese erhalten wir auf einem anderen Weg. Grundsätzlich finden wir Attribute nur bei Elementen des Typs **Element**. Hier können wir mit der Eigenschaft **HasAttributes** prüfen, ob überhaupt Attribute vorliegen. Ist das der Fall, lässt sich mit der Eigenschaft **AttributeCount** die Anzahl der Attribute ermitteln. Wir können diese aber auch direkt in einer **Do While**-Schleife durchlaufen.

Dazu verwenden wir wieder eine Abbruchbedingung, die gleichzeitig das nächste Attribut einliest. Diese lautet **MoveToNextAttribut**. Die Methode liefert den Wert **True**, wenn sie ein weiteres Attribut gefunden hat und **False**, wenn kein weiteres Attribut vorliegt. Den Namen des Attributs ermitteln Sie dann wiederum mit der Eigenschaft **Name**. Den Wert des Attributs liefert die Eigenschaft **Value**.

Bestellungen per XmlReader einlesen

Mit dem XmlReader durchlaufen Sie ein XML-Dokument Element für Element. Da dies recht unflexibel ist, müssen wir beim Einlesen verschachtelter XML-Dokumente umso flexibler programmieren. Dieser Artikel zeigt, wie Sie die Daten eines XML-Dokuments mit Kunden, Bestellungen, Bestellpositionen und Artikeln per XmlReader zuerst in ein Entity Data Model einlesen und die Daten von dort in den zugrunde liegenden Tabellen speichern.

Zu importierendes XML-Dokument

Das XML-Dokument, dessen Daten wir importieren wollen, hat ein Stammelement namens Bestellverwaltung. Darunter finden wir ein Auflistungselement namens **Kunden**, das alle **Kunde**-Elemente enthält. Das **Kunde**-Element hat ein Attribut zum Speichern der **ID** sowie einige untergeordnete Felder mit Firma, Vorname, Nachname und Adresse des Kunden:

```
<?xml version="1.0" encoding="utf-8"?>
<Bestellverwaltung>
  <Kunden>
    <Kunde ID="1">
      <Firma>André Minhorst Verlag</Firma>
      <Vorname>André</Vorname>
      <Nachname>Minhorst</Nachname>
      <Strasse>Borkhofer Str. 17</Strasse>
      <PLZ>47137</PLZ>
      <Ort>Duisburg</Ort>
```

Unterhalb des **Kunde**-Elements finden wir außerdem für jeden Kunden ein Auflistungselement namens **Bestellungen**. Dieses nimmt ein oder mehrere **Bestellung**-Elemente auf. Diese enthalten ebenfalls in einem Attribut namens **ID** einen eindeutigen Identifizierer. Darunter finden wir ein Element namens **Bestelldatum** mit dem Datum der Bestellung sowie ein weiteres Auflistungselement namens **Bestellpositionen**. Dieses enthält für jede Position ein Element des Typs **Bestellposition** mit dem Attribut **ID**.

Jede Bestellposition weist die vier Elemente **Artikel**, **Einzelpreis**, **Menge** und **Mehrwertsteuersatz** auf:

```
      <Bestellungen>
        <Bestellung ID="1">
          <Bestelldatum>2020-11-17T12:00:00+00:00</Bestelldatum>
          <Bestellpositionen>
            <Bestellposition ID="1">
              <Artikel>Access im Unternehmen</Artikel>
              <Einzelpreis>124</Einzelpreis>
              <Menge>1</Menge>
```

```
            <Mehrwertsteuersatz>7</Mehrwertsteuersatz>
          </Bestellposition>
```

Von den meisten dieser Elemente gibt es noch weitere Exemplare, die wir hier aus Platzgründen nicht explizit abbilden:

```
            ...
          </Bestellpositionen>
        </Bestellung>
        ...
      </Bestellungen>
    </Kunde>
    ...
  </Kunden>
</Bestellverwaltung>
```

Entity Framework vorbereiten

Nach dem Erstellen des WPF-Projekts mit Visual Basic fügen wir das Entity Framework hinzu. Dazu klicken Sie mit der rechten Maustaste auf das Projekt-Element im Projektmappen-Explorer und wählen aus dem Kontextmenü den Eintrag **Hinzufügen|Neues Element...** aus. Im Dialog **Neues Element hinzufügen** selektieren Sie **ADO.NET Entity Data Model** und nennen dieses **BestellverwaltungContext**.

Im folgenden Dialog **Assistent für Entity Data Model** wählen Sie die Option **Leeres Code First-Modell** aus. Dies legt die Elemente für das Entity Framework an und legt in der Datei **App.config** einen Connectionstring für eine Datenbank namens **Bestellverwaltung.BestellverwaltungContext** fest, wobei standardmäßig der Server **(LocalDb)\MSSQLLocalDB** verwendet wird.

Die Klasse **BestellverwaltungContext.vb** ergänzen wir wie folgt und definieren so die drei DbSet-Elemente, welche die Auflistungen für Kunden, Bestellungen und Bestellpositionen enthalten:

```
Public Class BestellverwaltungContext
    Inherits DbContext

    Public Sub New()
        MyBase.New("name=BestellverwaltungContext")
    End Sub

    Public Overridable Property Bestellpositionen() As DbSet(Of Bestellposition)
    Public Overridable Property Bestellungen() As DbSet(Of Bestellung)
    Public Overridable Property Kunden() As DbSet(Of Kunde)

End Class
```

Die Entitätsklasse für die Kunden hinterlegen wir wie folgt in der Klassendatei **Kunde.vb** in einem neuen Ordner namens **DataModel**. Die Klassen enthalten alle Informationen, die für das Erstellen in Form der Tabellen einer Datenbank notwendig sind. Dazu sind auch jeweils die **DataAnnotations**-Namespaces erforderlich:

```
Imports System.ComponentModel.DataAnnotations
Imports System.ComponentModel.DataAnnotations.Schema

<Table("Kunden")>
Public Partial Class Kunde
    Public Property ID As System.Int32
    <StringLength(255)>
    Public Property Firma As System.String
    <StringLength(255)>
    Public Property Vorname As System.String
    <StringLength(255)>
    Public Property Nachname As System.String
    <StringLength(255)>
    Public Property Strasse As System.String
    <StringLength(255)>
    Public Property PLZ As System.String
    <StringLength(255)>
    Public Property Ort As System.String
    Public Overridable Property Bestellungen As ICollection(Of Bestellung)
End Class
```

Die Klasse für die Bestellpositionen heißt **Bestellung.vb** und enthält die folgende Beschreibung der Felder:

```
Imports System.ComponentModel.DataAnnotations
Imports System.ComponentModel.DataAnnotations.Schema

<Table("Bestellungen")>
Public Partial Class Bestellung
    Public Property ID As System.Int32
    Public Property KundeID As System.Int32
    <Column(TypeName:="datetime2")>
    Public Property Bestelldatum As System.DateTime
    Public Overridable Property Bestellpositionen As ICollection(Of Bestellposition)
    Public Overridable Property Kunde As Kunde
End Class
```

Fehlt noch die Klasse für die Bestellpositionen, die wir **Bestellposition.vb** nennen:

```
Imports System.ComponentModel.DataAnnotations
Imports System.ComponentModel.DataAnnotations.Schema

<Table("Bestellpositionen")>
Public Partial Class Bestellposition
    Public Property ID As System.Int32
    <StringLength(255)>
    Public Property Artikel As System.String
    Public Property Einzelpreis As System.Decimal
    <StringLength(255)>
    Public Property Menge As System.String
    Public Property Mehrwertsteuersatz As System.Decimal
    Public Property BestellungID As System.Int32
    Public Overridable Property Bestellung As Bestellung
End Class
```

Datenbank anlegen

Zeigen Sie nun die Paket-Manager-Konsole an (Menüeintrag **Ansicht|Weitere Fenster|Paket-Manager-Konsole**) und geben Sie dort nacheinander die drei folgenden Befehle ein:

```
enable-migrations
add-migration init
update-database
```

Damit erstellen Sie die Datenbank auf Basis der Klassen und DbSets des Entity Data Models in der Datenbank, die in der Verbindungszeichenfolge angegeben ist.

System.XML-Namespace verfügbar machen

In der XAML-Datei **MainWindow.xaml** wollen wir das Einlesen mit einer Schaltfläche starten. Diese deklarieren wir wie folgt innerhalb des **Grid**-Elements:

```
<Button x:Name="btnKundenUndBestellungenInKlassen" Click="btnKundenUndBestellungenInKlassen_Click">Kunden und Bestellungen in Klassen</Button>
```

In der Code behind-Klasse **MainWindow.xaml.vb** hinterlegen wir einen Verweis auf den Namespace **System.Xml**:

```
Imports System.Xml
```

Klasse initialisieren per Konstruktor

Der Klasse fügen wir eine Variable namens **dbContext** hinzu, mit der wir auf den Datenbankkontext **BestellverwaltungContext** zugreifen:

XML-Dokumente erstellen mit XmlWriter

Im Artikel »XML-Dokumente schnell lesen mit XmlReader« haben Sie die Klasse XmlReader kennengelernt, mit der Sie schnell durch XML-Dokumente navigieren können. Damit können Sie zum Beispiel Exporte aus anderen Anwendungen in Ihre .NET-Anwendung importieren. Aber was, wenn Sie selbst einmal einen Export mit Ihren Daten im XML-Format bereitstellen wollen? Dazu gibt es eine passende Klasse namens XmlWriter. Wie Sie damit XML-Dokumente erstellen und mit den gewünschten Daten füllen, erfahren Sie im vorliegenden Artikel.

Namespace System.Xml referenzieren

Genau wie die Klasse **XmlReader** ist auch **XmlWriter** Bestandteil des Namespaces **System.Xml**. Diesen referenzieren wir deshalb wie folgt im Kopf der Code behind-Klasse unseres Beispielfensters **MainWindow.xaml**:

```
Imports System.Xml
```

Leeres XML-Dokument erstellen mit Create

Um ein XML-Dokument als Datei zu erstellen, rufen Sie die Methode **Create** des **XmlWriter**-Objekts auf und übergeben dieser den Pfad der zu erstellenden XML-Datei. Wenn Sie einfach nur den Dateinamen angeben, wird die Datei im gleichen Verzeichnis erstellt wie die **.exe**-Datei des Projekts:

```
Dim objWriter As XmlWriter
objWriter = XmlWriter.Create("XmlWriter.xml")
objWriter.Close()
```

Dies legt allerdings nur ein leeres Dokument an.

XML-Dokument mit Element erstellen

Um ein Dokument mit dem Prolog und einem ersten Element zu erstellen, bedarf es einiger weiterer Zeilen. Die Methoden **WriteStartDocument** und **WriteEndDokument** starten den Schreibprozess und beenden diesen wieder. Die Methode **WriteStartElement** fügt ein Starttag namens **Bestellverwaltung** hinzu und **WriteEndElement** schließt das Element mit dem Endtag ab. Dabei müssen Sie **WriteEndElement** keinen Parameter mehr übergeben – die Methode bezieht sich immer auf das zuletzt geöffnete Starttag:

```
Private Sub btnDokumentMitElementErstellen_Click(sender As Object, e As RoutedEventArgs)
    Dim objWriter As XmlWriter
    objWriter = XmlWriter.Create("XmlWriter.xml", objSettings)
    objWriter.WriteStartDocument()
    objWriter.WriteStartElement("Bestellverwaltung")
    objWriter.WriteEndElement()
```

```
    objWriter.WriteEndDocument()
    objWriter.Close()
End Sub
```

Das Ergebnis ist das folgende XML-Dokument:

```
<?xml version="1.0" encoding="utf-8"?><Bestellverwaltung />
```

Hier finden wir keinen Zeilenumbruch, was wir als Erstes ändern möchten

Eigenschaften für das Erstellen festlegen

Die **Create**-Methode erwartet in einer Überladung einen zweiten Parameter des Typs **XmlWriterSettings**. Wir erstellen also zuvor ein solches Objekt und übergeben es dann – allerdings nicht, ohne einige Eigenschaften dieser Klasse zu nutzen:

- **Encoding**: Gibt an, ob das Dokument mit UTF8, Unicode oder einer anderen Kodierung erstellt werden soll. Die verschiedenen Werte stellt **System.Text.Encoding** zur Verfügung, was Sie auch per IntelliSense nutzen können. Standardwert ist **UTF8**.

- **Indent**: Boolean-Wert, der angibt, ob Einrückungen hinzugefügt werden sollen.

- **IndentChar**: String-Wert, der die für einen Einrückungsschritt zu verwendende Zeichenkette enthält. Wenn Sie das Tabulatorzeichen verwenden wollen, geben Sie etwa **ControlChars.Tab** an, wenn Sie eine bestimmte Anzahl Leerzeichen verwenden wollen, nutzen Sie einfach die Zeichenkette **" "**.

- **NewLineChars**: Gibt an, welches Zeichen für Zeilenumbrüche verwendet werden soll (Standard: **Chr(13) + Chr(10)**).

- **NewLineOnAttributes**: Gibt an, ob Attribute in einer eigenen Zeile ausgegeben werden sollen. Dient bei vielen Attributen gegebenenfalls der Übersichtlichkeit beim manuellen Lesen.

- **OmitXmlDeclaration**: Gibt mit **True** an, ob der Prolog weggelassen werden soll.

Im folgenden Beispiel nutzen wir die beiden Eigenschaften **Indent** und **IndentChar**, um festzulegen, dass jeder Einzug durch vier Leerzeichen realisiert werden soll. Außerdem haben wir ein paar weitere Elemente unterhalb des **Bestellverwaltung**-Elements hinzufügt, damit die Einzüge im erstellten Dokument zur Geltung kommen:

```
Private Sub btnDokumentMitEinstellungen_Click(sender As Object, e As RoutedEventArgs)
    Dim objWriter As XmlWriter
    Dim objSettings As XmlWriterSettings
    objSettings = New XmlWriterSettings
    With objSettings
        .Indent = True
        .IndentChars = "    "
```

```
    End With
    objWriter = XmlWriter.Create("XmlWriter.xml", objSettings)
    objWriter.WriteStartDocument()
    objWriter.WriteStartElement("Bestellverwaltung")
    objWriter.WriteStartElement("Kunden")
    objWriter.WriteEndElement()
    objWriter.WriteEndElement()
    objWriter.WriteEndDocument()
    objWriter.Close()
End Sub
```

Das Ergebnis sieht nun wie folgt aus:

```
<?xml version="1.0" encoding="utf-8"?>
<Bestellverwaltung>
    <Kunden>
        <Kunde />
    </Kunden>
</Bestellverwaltung>
```

Mehrere Elemente auf einer Ebene

Bisher haben wir mit **WriteStartElement**/**WriteEndElement** immer Elemente erstellt, die unter einem anderen Element angeordnet werden. Wenn Sie beispielsweise zwei **Kunde**-Elemente im Element **Kunden** anlegen wollen, müssen Sie einfach das erste **Kunde**-Element abschließen und dann das nächste **Kunde**-Element beginnen:

```
---
objWriter.WriteStartElement("Kunden")
objWriter.WriteStartElement("Kunde")
objWriter.WriteEndElement()
objWriter.WriteStartElement("Kunde")
objWriter.WriteEndElement()
objWriter.WriteStartElement("Kunde")
objWriter.WriteEndElement()
objWriter.WriteEndElement()
---
```

Elemente mit Inhalt erstellen mit WriteElementString

Wenn Sie dem **Kunde**-Element nun Elemente wie **Firma**, **Vorname** und **Nachname** mit den entsprechenden Werten hinzufügen wollen, verwenden Sie nicht die **WriteStartElement**-Methode, sondern die **WriteElementString**-Methode. Diese erwartet als ersten Parameter den Namen des zu erstellenden Elements und als zweiten Parameter den Wert. Wenn wir dies nach dem Erstellen des **Kunde**-Starttags erledigen, werden **Firma**, **Vorname** und **Nachname** wie folgt dem Element **Kunde** untergeordnet:

Bestellungen per XmlWriter in XML exportieren

Im Artikel »XML-Dokumente erstellen mit XmlWriter« haben wir die grundlegenden Techniken beschrieben, mit denen Sie XML-Dokumente mit der XmlWriter-Klasse erzeugen und mit den gewünschten Daten füllen. Im vorliegenden Artikel liefern wir ein Praxisbeispiel dazu und wollen die Daten eines Entity Data Models, das aus einer SQL Server-Datenbank befüllt wird, in ein XML-Dokument schreiben. Dieses können Sie dann mit der Lösung aus dem Artikel »Bestellungen per XmlReader einlesen« wieder einlesen und somit Daten zwischen zwei Anwendungen per XML-Dokument übertragen.

Im Artikel **Bestellungen per XmlReader einlesen** (**www.datenbankentwickler.net/234**) haben wir gezeigt, wie Sie Daten aus einem XML-Dokument in ein Entity Data Model einlesen und diese dann in der damit verknüpften SQL Server-Datenbank speichern. Im vorliegenden Artikel wollen wir nun den umgekehrten Weg gehen. Dazu nutzen wir das im oben genannten Artikel erstellte Projekt. Dort haben wir schon beschrieben, wie das Entity Data Model, bestehend aus den drei Entitäten **Kunde**, **Bestellung** und **Bestelldetail** aufgebaut ist. Darüber wollen wir nun die Daten der dahinter stehenden Tabellen in das Entity Data Model einlesen und diese dann in ein neues XML-Dokument exportieren.

Damit lernen Sie nicht nur die Möglichkeiten der **XmlWriter**-Klasse besser kennen, sondern es ist auch eine schöne Fingerübung für den Umgang mit den Daten eines Entity Data Models.

Dokument erstellen und Schreiben der Daten initialisieren

Die Schaltfläche **btnKundenUndBestellungenInXMLDokument** löst die folgende Methode aus. Diese deklariert neben einer Variablen für die Zieldatei noch ein Listenelement für die Kunden sowie ein **XmlWriter**- und ein **XmlWriterSettings**-Objekt. Letzteres erstellt die Methode zuerst und legt fest, dass wir Einrückungen mit vier Leerzeichen verwenden wollen:

```
Private Sub btnKundenUndBestellungenInXMLDokument_Click(sender As Object, e As RoutedEventArgs)
    Dim strXmlDatei As String
    Dim Kunden As List(Of Kunde)
    Dim objWriter As XmlWriter
    Dim objSettings As XmlWriterSettings
    objSettings = New XmlWriterSettings
    With objSettings
        .Indent = True
        .IndentChars = "    "
    End With
```

Danach ermitteln wir mit der Funktion **ZieldateiErmitteln** den Pfad zu der zu erstellenden Datei und erstellen unter Angabe dieses Pfades mit der **Create**-Methode das **XmlWriter**-Objekt:

```
    strXmlDatei = ZieldateiErmitteln()
```

```
    objWriter = XmlWriter.Create(strXmlDatei, objSettings)
```

Nach dem notwendigen Aufruf der **WriteStartDocument**-Methode füllen wir alle Kunden der Tabelle **Kunden** über das **DbSet** namens **Kunden** des Entity Data Models in die Liste **Kunden**. Damit und mit **objWriter** als Parameter rufen wir eine weitere Methode namens **KundenInXMLSchreiben** auf, welche die Kunden zum XML-Dokument hinzufügen soll:

```
    objWriter.WriteStartDocument()
    Kunden = New List(Of Kunde)(dbContext.Kunden)
    KundenInXMLSchreiben(objWriter, Kunden)
    objWriter.WriteEndElement()
    objWriter.WriteEndDocument()
    objWriter.Close()
End Sub
```

Zielpfad mit SaveFileDialog auswählen

Die oben verwendete Funktion **ZieldateiErmitteln** verwendet die **SaveFileDialog**-Klasse, um den Zielpfad zu ermitteln. Dazu stellen wir als Standard-Dateiendung den Wert **.xml**, als Filter ebenfalls **.xml** und als Startverzeichnis das Verzeichnis der **.exe**-Datei der Anwendung ein. Die **ShowDialog**-Methode öffnet den Dialog und der Code läuft erst weiter, wenn dieser geschlossen wird. Liefert **ShowDialog** den Wert **True**, wurde eine Datei ausgewählt. Diese speichert die Methdoe in **strZieldatei** gespeichert und gibt den enthalten Wert dann mit **Return** an die aufrufende Routine zurück:

```
Private Function ZieldateiErmitteln()
    Dim objFileDialog As SaveFileDialog
    Dim strZieldatei As String
    objFileDialog = New SaveFileDialog
    With objFileDialog
        .DefaultExt = ".xml"
        .Filter = "XML-Dokumente (.xml)|*.xml"
        .InitialDirectory = System.AppDomain.CurrentDomain.BaseDirectory
    End With
    If (objFileDialog.ShowDialog) = True Then
        strZieldatei = objFileDialog.FileName
    End If
    Return strZieldatei
End Function
```

Kunden in die XML-Datei schreiben

Die von der Hauptmethode aufgerufene Routine KundenInXMLSchreiben erwartet zwei Parameter: das **XmlWriter**-Objekt aus **objWriter** sowie die dort ermittelte Liste der Kunden aus der Datenbank. Wichtig ist, dass Sie den Parameter **objWriter** mit dem Schlüsselwort **ByRef** auszeichnen. So wirken sich die Änderungen an **objWriter** aus dieser Routine auch auf die Variable aus, die von der aufrufenden Methode aus übergeben wurde.

Die Routine **KundenInXMLSchreiben** erstellt en neues **Kunden**-Element in **objWriter** und durchläuft dann in einer **For Each**-Schleife alle Elemente der Auflistung Kunden. Dabei legt sie unter dem **Kunden**-Element jeweils ein neues **Kunde**-Element an. Das **Kunde**-Element erhält dann ein Attribut mit dem Namen **ID** und dem Primärschlüsselwert des Kunden als Wert. Dann fügt die Routine weitere Elemente mit den Inhalten der Eigenschaften **Firma**, **Vorname**, **Nachname**, **Strasse**, **PLZ** und **Ort** hinzu. Schließlich liest sie alle **Bestellung**-Elemente des Kunden in die Liste **Bestellungen** ein. Diese übergibt sie neben **objWriter** an eine weitere Methode, die für jeden Kunden einmal aufgerufen wird und die **BestellungenInXMLSchreiben** heißt:

```
Private Sub KundenInXMLSchreiben(ByVal objWriter As XmlWriter, Kunden As List(Of Kunde))
    Dim objKunde As Kunde
    Dim Bestellungen As List(Of Bestellung)
    objWriter.WriteStartElement("Kunden")
    For Each objKunde In Kunden
        objWriter.WriteStartElement("Kunde")
        objWriter.WriteAttributeString("ID", objKunde.ID)
        objWriter.WriteElementString("Firma", objKunde.Firma)
        objWriter.WriteElementString("Vorname", objKunde.Vorname)
        objWriter.WriteElementString("Nachname", objKunde.Nachname)
        objWriter.WriteElementString("Strasse", objKunde.Strasse)
        objWriter.WriteElementString("PLZ", objKunde.PLZ)
        objWriter.WriteElementString("Ort", objKunde.Ort)
        Bestellungen = New List(Of Bestellung)(From b In dbContext.Bestellungen
                                               Where b.Kunde.ID = objKunde.ID)
        BestellungenInXMLSchreiben(objWriter, Bestellungen)
        objWriter.WriteEndElement()
    Next
    objWriter.WriteEndElement()
End Sub
```

Mit dieser ersten Methode haben wir bereits die Grundstruktur des XML-Dokuments hergestellt. Was nun noch fehlt, sind die Bestellungen der Kunden:

```
<?xml version="1.0" encoding="utf-8"?>
<Kunden>
    <Kunde ID="29">
        <Firma>André Minhorst Verlag</Firma>
        <Vorname>André</Vorname>
        <Nachname>Minhorst</Nachname>
        <Strasse>Borkhofer Str. 17</Strasse>
        <PLZ>47137</PLZ>
        <Ort>Duisburg</Ort>
        ... Bestellungen
```

XML lesen mit dem Document Object Model

Mit der Klasse XmlReader, die wir im Artikel »XML-Dokumente schnell lesen mit XmlReader« vorgestellt haben, können Sie XML-Dokumente schnell sequenziell durchlaufen. Dies eignet sich vor allem für den Zugriff auf sehr große XML-Dokumente. Wenn Sie selektiver auf die Inhalte des XML-Dokuments zugreifen wollen, können Sie das Document Object Model nutzen. Implementierungen dieser Schnittstelle, die kurz DOM genannt wird, gibt es für fast alle Programmiersprachen – so auch für die von .NET. Dieser Artikel zeigt, wie Sie die Klassen, Eigenschaften und Methoden des Document Object Models für den Zugriff auf XML-Dokumente nutzen.

DOM im Gegensatz zu XmlReader und XmlWriter

Einer der wichtigsten Unterschiede des Document Object Model im Vergleich zur **XmlReader**-Klasse ist, dass Sie damit nicht nur lesend, sondern auch schreibend auf die XML-Dokumente zugreifen können. Das heißt, wenn Sie ein Element gefunden haben, dessen Inhalt Sie ändern möchten, dann können Sie das direkt mit den Methoden des Document Object Models erledigen. Sie können damit nicht nur Elemente bearbeiten, sondern auch neue Elemente hinzufügen oder bestehende Elemente löschen. Und natürlich können Sie damit auch komplett neue XML-Dokumente erstellen und mit Daten füllen. Dazu steht allerdings auch die Klasse **XmlWriter** zur Verfügung, die wir im Artikel **XML-Dokumente erstellen mit XmlWriter** vorgestellt haben.

Mit dem Document Object Model laden Sie das komplette XML-Dokument in den Speicher, weshalb einzelne Elemente gezielt angesteuert werden können. Während beim XmlReader für alle Elemente die gleichen Eigenschaften abgefragt werden können und je nach Elementtyp mal die eine, mal die andere Eigenschaft Werte zurückliefert, liest das Document Object Model die einzelnen Elemente des XML-Dokuments mit einem entsprechenden Objekttyp ein.

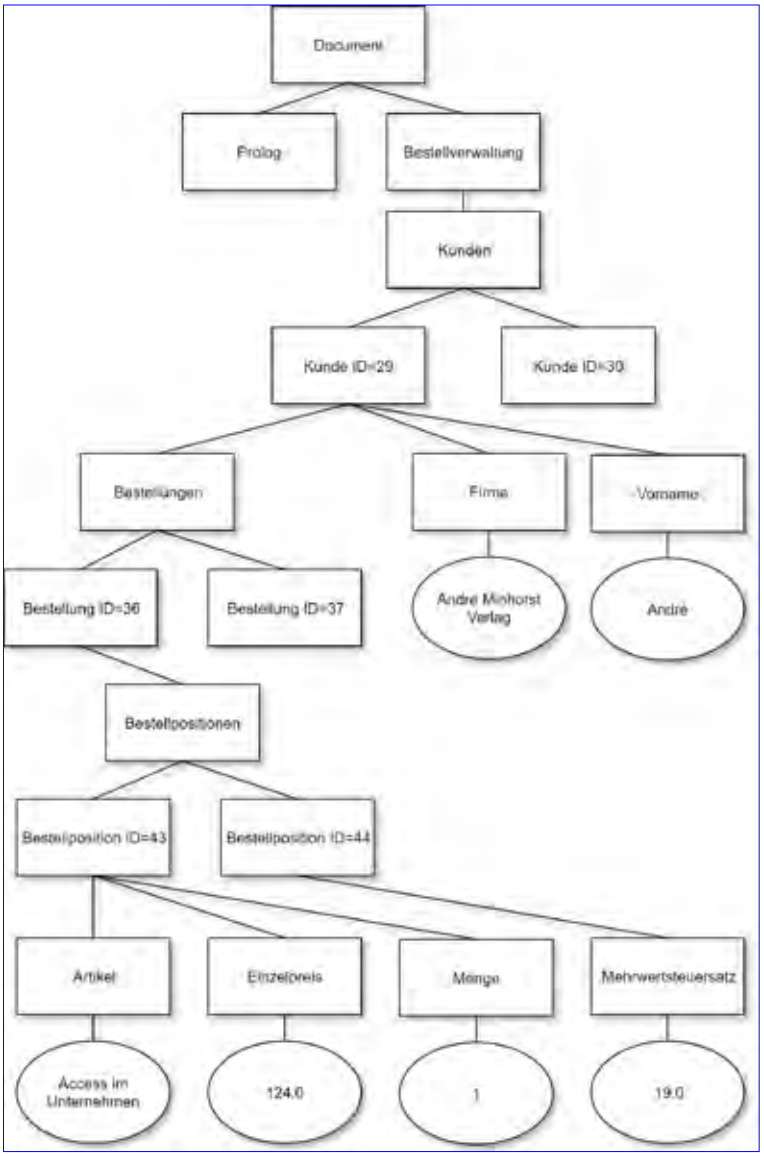


Bild 1: Baum eines XML-Dokuments

XML-Baum

Intern bildet DOM das XML-Dokument als Baumstruktur ab. Das kann man sich so vorstellen, dass wir ein Stammelement haben, das an der Spitze sitzt. Darunter gibt es dann ein oder mehrere untergeordnete Elemente, die wiederum untergeordnete Elemente haben, bis dann irgendwann Werte in den Elementen stehen – diese landen immer am anderen Ende des Baums. Einen Baum für unser Beispiel-XML-Dokument finden Sie in Bild 1.

Dieser bildet die aus Platzgründen gekürzte Version des XML-Dokuments aus Bild 2 ab.

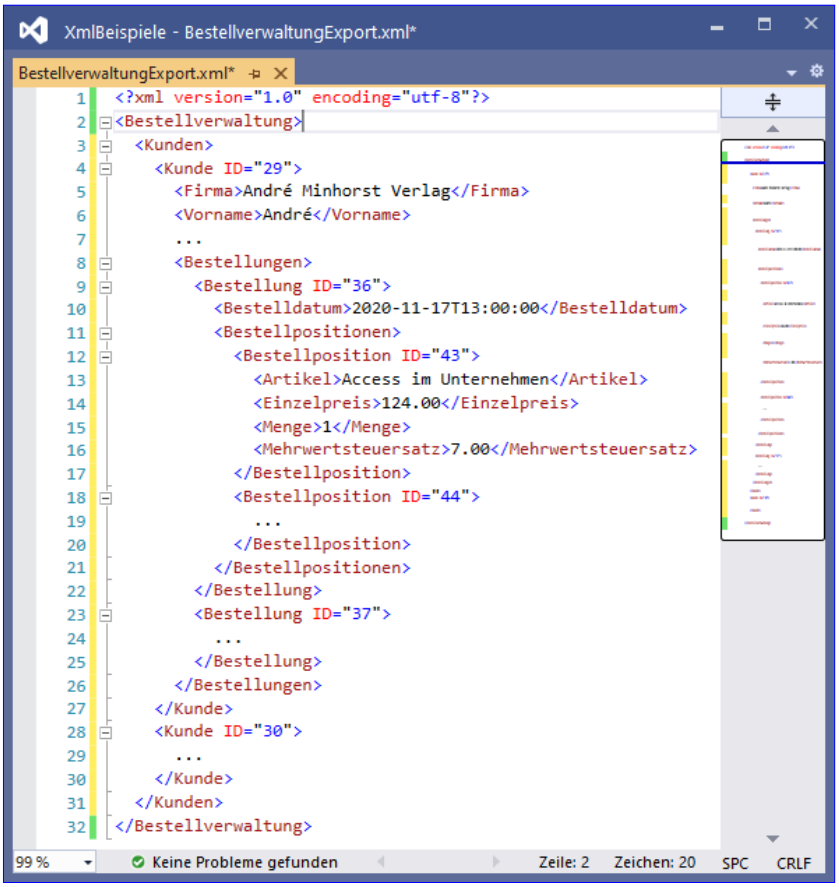


Bild 2: Ausschnitt des Dokuments entsprechend dem Baum aus der ersten Abbildung

Verweis auf den Namespace System.Xml

Bevor wir ein Objekt auf Basis der Klasse **XmlDocument** nutzen können, müssen wir einen Verweis auf den folgenden Namespace zur Klasse hinzufügen:

```
Imports System.Xml
```

Deklarieren und Initialisieren von XmlDocument

Mit den folgenden beiden Anweisungen deklarieren und initialisieren wir ein Objekt namens **objXML** mit dem Typ **XmlDocument**:

```
Dim objXML As XmlDocument
objXML = New XmlDocument
```

Laden des XML-Dokuments

Für das Laden eines XML-Dokuments bietet die **XmlDocument**-Klasse zwei Methoden an:

- **Load**: Erwartet den Pfad des zu ladenden XML-Dokuments als Parameter, falls es sich um eine Datei handelt. Sie können auch eine URL angeben. Überladungen erlauben auch das Laden aus einem **Stream**-, einem **TextReader**- oder einem **XmlReader**-Objekt.

- **LoadXml**: Erwartet das XML-Dokument als String-Parameter.

Die folgenden Anweisungen erzeugen ein **XmlDocument**-Objekt, laden das Dokument **Bestellverwaltung.xml** hinein und geben den Inhalt über die Eigenschaft **OuterXml** aus:

```
Dim objXML As XmlDocument
objXML = New XmlDocument
objXML.Load("C:\...\Bestellverwaltung.xml")
Debug.Print(objXML.OuterXml)
```

Die Ausgabe mit **InnerXml** liefert für das **XmlDocument** selbst übrigens das gleiche Ergebnis wie **OuterXml**.

Fehler beim Einlesen abfangen

Im Gegensatz zur **Load**-Methode von DOM unter VBA, die einen **Boolean**-Wert zurückliefert, falls beim Einlesen ein Fehler auftritt, hat die **Load**-Methode unter VB.NET keinen Rückgabewert. Wenn ein nicht gültiges XML-Dokument geladen wird, köst die Methode schlicht und einfach einen Fehler aus. Diesen können wir jedoch per **Try...Catch** wie folgt abfangen:

```
Try
    objXML.Load("C:\...\Bestellverwaltung.xml")
Catch ex As XmlException
    MessageBox.Show("Fehler beim Einlesen:" + vbNewLine + vbNewLine + ex.Message)
End Try
```

Sie können einen solchen Fehler leicht auslösen, indem Sie einen der Start- oder Endtags aus der Beispieldatei **Bestellverwaltung.xml** entfernen und dann versuchen, diese einzulesen.

Klassen für die Untersuchung von XML-Dokumenten mit DOM

Die Basisklasse für die Arbeit mit XML-Dokumenten heißt **XmlNode**. Es gibt einige abgeleitete Klassen, zu denen auch **XmlDocument** gehört. Das heißt, das **XmlNode** die grundlegenden Methoden und Eigenschaften bereitstellt und die abgeleiteten Klassen je nach Klasse weitere Elemente. **XmlDocument** bietet beispielsweise die bereits beschriebenen Methoden **Load** und **LoadXml** an. Weitere von **XmlNode** abgeleitete Klassen sind **XmlLinkedNode** und **XmlAttribute**. Von **XmlLinkedNode** sind weiterhin beispielsweise **XmlProcessingInstruction**, **XmlDeclaration** oder **XmlElement** abgeleitet – sowie die Klasse **XmlCharacterData**, die wiederum die Basis für Klassen wie **XmlComment** oder **XmlCDATASection** bildet.

Das heißt, dass Sie alle Elemente eines XML-Dokuments mit der Klasse **XmlNode** referenzieren können, aber wenn Sie die spezifischen Methoden und Eigenschaften nutzen wollen, finden Sie den genauen Typ heraus und nutzen diesen.

Nodes durchlaufen mit der ChildNodes-Auflistung

Mit **objXML** haben wir nun einen Verweis auf das **XmlDocument**-Objekt selbst. Wie kommen wir nun an die untergeordneten Elemente heran? Dafür bietet die **XmlNode**-Klasse und somit auch die **XmlDocument**-Klasse die Auflistung **ChildNodes** an. Wir können zunächst herausfinden, ob **XmlDocument** überhaupt untergeordnete Elemente hat. Das erledigen wir mit der Eigenschaft **HasChildNodes**. Sind untergeordnete Elemente vorhanden, können wir über die **ChildNodes**-Auflistung auf diese zugreifen. Diese bietet mit der Eigenschaft **Count** die Möglichkeit, die Anzahl zu ermitteln. Im folgenden Beispiel geben wir die

Werte von **HasChildNodes** und **ChildNodes.Count** aus und durchlaufen dann eine Schleife über alle **ChildNode**-Elemente. Diese referenzieren wir mit einer Variablen des Typs **XmlNode**. Das ist wichtig, weil sich in der Auflistung **ChildNodes** verschiedene Elemente mit von **XmlNode** abgeleiteten Klassen befinden. Wenn wir eine Variable des Typs **XmlElement** nutzen, erhalten wir einen Fehler, wenn eines der **ChildNode**-Elemente etwa den Typ **XmlDeclaration** hat.

Wir können aber innerhalb der Schleife genau herausfinden, um welchen Typ es sich handelt. Dazu nutzen wir die Eigenschaft **NodeType**. Außerdem geben wir mit folgenden Zeilen den Namen der untergeordneten Elemente aus:

```
Dim objNode As XmlNode
Debug.Print (objXML.HasChildNodes)
Debug.Print (objXML.ChildNodes.Count)
For Each objNode In objXML.ChildNodes
    Debug.Print (objNode.Name + " " + objNode.NodeType.ToString)
Next objNode
```

Das Ergebnis für dieses Beispiel lautet:

```
xml XmlDeclaration
Bestellverwaltung Element
```

Damit haben wir also die beiden Elemente der obersten Ebene erfasst, wobei das erste den Typ **XmlDeclaration** und das zweite den Typ **Element** hat:

```
<?xml version="1.0" encoding="utf-8"?>
<Bestellverwaltung>
```

Nun wollen wir auch die übrigen Elemente erkunden. Dazu erstellen wir eine rekursiv definierte Methode, mit der wir uns durch die Hierarchie des XML-Dokuments arbeiten.

Nodes rekursiv durchlaufen

Die rekursive Methode rufen wir in der Schleife der Hauptmethode wie folgt auf:

```
For Each objNode In objXML.ChildNodes
    Debug.Print(objNode.Name + " " + objNode.NodeType.ToString)
    Nodes_Rekursiv(objNode, "  ")
Next objNode
```

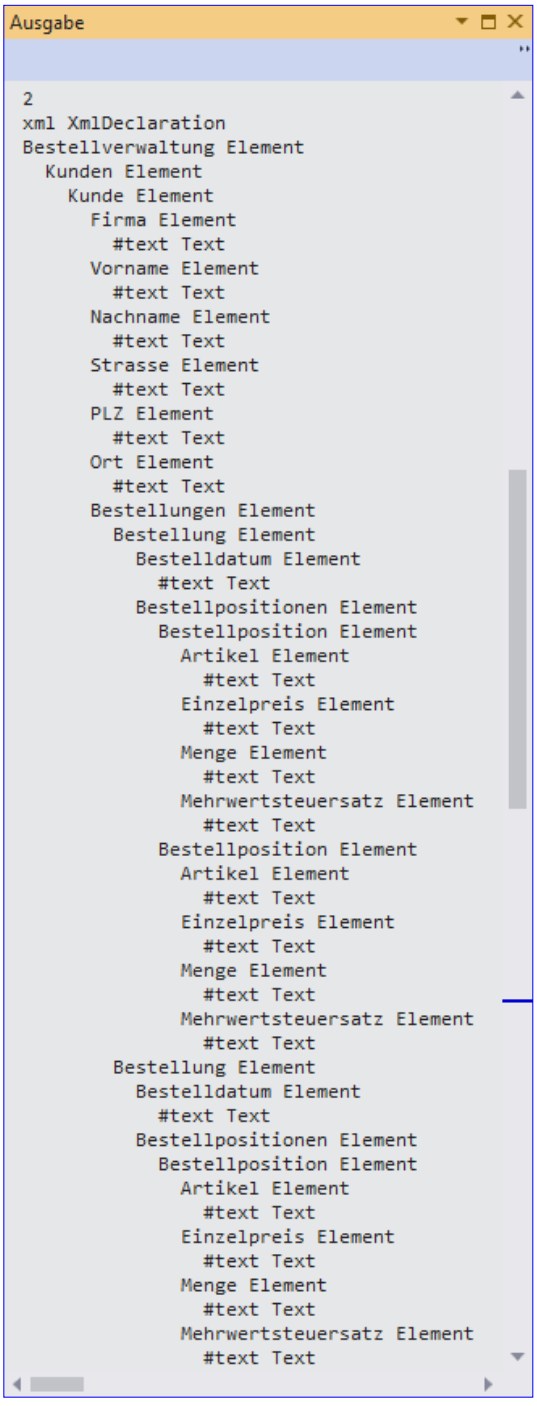


Bild 3: Name und Typen von Elementen

Onlinebanking mit DDBAC: Benutzeroberfläche

Nachdem wir die Funktionen zum Abrufen von Kontoständen und Umsätzen sowie für das Überweisen programmiert haben, können wir uns an den Entwurf einer Benutzeroberfläche für diese Funktionen begeben. Diese soll kompakte Formulare für die wichtigsten bereits behandelten Onlinebanking-Funktionen enthalten. Dazu gehört die Auswahl von Bankaccount und Konto sowie die Anzeige von Umsätzen und Kontostand. Dies erledigen wir im Hauptfenster der Anwendung. Ein weiteres Fenster erlaubt dann das Ausführen einer Überweisung für das aktuell im Hauptfenster ausgewählte Konto. Außerdem benötigen wir noch eine Tabelle, mit der wir die abgefragten Umsatzdaten speichern können. Diese fügen wir per Entity Framework hinzu.

Voraussetzungen

Voraussetzung für das Umsetzen der Lösung dieses Artikels ist das Vorhandensein einer DDBAC-Lizenz. Die damit nutzbaren Komponenten sind leider nicht mehr kostenlos verfügbar, sondern müssen lizensiert werden, bevor man diese in seine Produkte einbauen kann. Lizenzen finden Sie im Onlineshop unter **https://shop.minhorst.com/access-tools/295/ddbac-jahreslizenz?c=78**.

Vorbereitungen

Als Projekt legen wir ein neues Projekt des Typs **Visual Basic|Windows Desktop|WPF-App** an.

Dann fügen wir ein neues Element des Typs **ADO.NET Entity Data Model** hinzu. Dieses dient dazu, ein Entity Data Model zu erstellen und über dieses automatisch eine Datenbank mit der benötigten Tabelle zu generieren. Nachdem Sie mit dem Menübefehl **Projekt|Neues Element hinzufügen...** den Dialog **Neues Element hinzufügen** geöffnet haben, tragen Sie den Namen des Entity Data Models ein (hier **OnlinebankingContext**) und klicken auf **Hinzufügen**. Im nun erscheinenden **Assistent für Entity Data Model** wählen Sie den Typ **Leeres Code First-Modell** aus. Wenige Sekunden später hat der Assistent die notwendigen Elemente hinzugefügt.

Durch diesen Schritt haben wir außerdem direkt Änderungen an der Datei **App.config** vorgenommen. Dort steht nun im Element **connectionStrings** die Verbindungszeichenfolge, mit der wir später die Datenbank zum Speichern der Umsätze erstellen und verwenden.

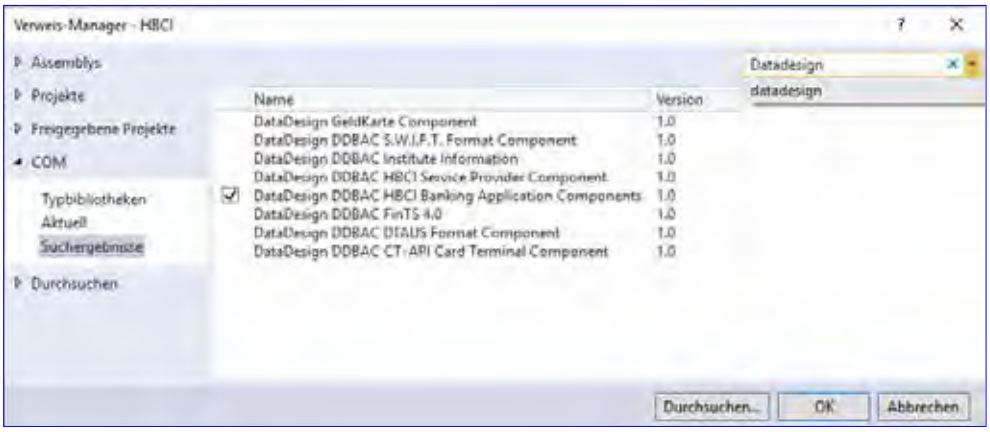


Bild 1: Verweis auf die DDBAC-Bibliothek

Außerdem benötigen wir noch eine Referenz auf die

DDBAC-Komponenten, die den Zugriff auf Ihre Konten überhaupt erst ermöglichen. Dazu fügen Sie dem Projekt den Verweis aus Bild 1 hinzu.

Kontostand anzeigen

Wir beginnen mit der Anzeige des Kontostands. Hier wollen wir in einem Fenster die Auswahl des Kontakts und anschließend des gewünschten Kontos erlauben. Per Klick auf eine Schaltfläche soll dann der Kontostand eingelesen und in einem Textfeld angezeigt werden. Wie Sie bereits in den vorherigen Artikeln erfahren haben, legen wir die Kontakte und Konten über die Systemsteuerung an, und zwar über die App **Homebanking Administrator (32-Bit)**. Auf die dort angelegten und synchronisierten Elemente können wir dann per Code zugreifen.

Ein neues WPF-Fenster soll nun zunächst ein Kombinationsfeld zur Auswahl des Kontakts anbieten. Nachdem der Benutzer dort einen Eintrag ausgewählt hat, sollen die zum gewählten Kontakt gehörenden Konten in einem zweiten Kombinationsfeld angezeigt werden. Das passende Steuerelement für diesen Zweck ist jeweils das **ComboBox**-Element. Also fügen wir zunächst zwei solche Elemente im Hauptfenster **MainWindow.xaml** an. Damit die zusätzlich angelegten Bezeichnungsfelder optisch ansprechend dargestellt werden, erstellen wir zuvor einige Spalten und Zeilen im bereits vorhandenen **Grid**-Element.

Klasse für die Kontakte anlegen

Die über die Systemsteuerung hinzufügten Onlinebanking-Kontakte lesen wir mithilfe der im Artikel **Onlinebanking mit DDBAC: Saldo und Umsätze** beschriebenen Funktion ein, die wir jedoch leicht abwandeln. Die eingelesenen Daten sollen je Kontakt in einer neuen Instanz der nachfolgend beschriebenen Klasse landen:

```
Public Class Contact
    Public BankCode As String
    Public CountryCode As String
    Public UserID As String
End Class
```

Diese Klasse legen Sie als neues Element im Projekt an und speichern es unter **Contact.vb**. Wir benötigen allerdings die **INotifyPropertyChanged**-Schnittstelle in dieser Klasse, daher ändern wir diese wie folgt.

Wir fügen außerdem noch eine Eigenschaft namens **BACContact** hinzu, mit der wir einen Verweis auf das **BACContact**-Objekt speichern, aus dem wir die Eigenschaften der Klasse ausgelesen haben:

```
Imports System.ComponentModel

Public Class Contact
    Implements INotifyPropertyChanged
    Public Event PropertyChanged As PropertyChangedEventHandler Implements INotifyPropertyChanged.PropertyChanged
    Protected Overridable Sub OnPropertyChanged(propname As String)
        RaiseEvent PropertyChanged(Me, New PropertyChangedEventArgs(propname))
    End Sub
```

```
Private _bankCode As String
Private _countryCode As String
Private _userID As String
Private _bACContact As BACContact

Public Property BankCode As String
    Get
        Return _bankCode
    End Get
    Set(value As String)
        _bankCode = value
        OnPropertyChanged("BankCode")
    End Set
End Property

Public Property CountryCode As String
    Get
        Return _countryCode
    End Get
    Set(value As String)
        _countryCode = value
        OnPropertyChanged(CountryCode)
    End Set
End Property

Public Property UserID As String
    Get
        Return _userID
    End Get
    Set(value As String)
        _userID = value
        OnPropertyChanged("UserID")
    End Set
End Property

Public Property BACContact As BACContact
    Get
        Return _bACContact
    End Get
    Set(value As BACContact)
        _bACContact = value
```

```
        End Set
    End Property
End Class
```

Klasse für die Konten anlegen

Die Klasse für die Konten sieht ähnlich aus wie die für die Kontakte. Auch hier implementieren wir die **INotifyPropertyChanges**-Schnittstelle. Die Klasse bietet die Eigenschaften **AccountNumber**, **AcctName**, **BIC** und **IBAN** an. Darüberhinaus wollen wir das **BACAccount**-Objekt, über das wir die Klasse füllen, mit den Instanzen der Klasse speichern. Die Klassendefinition sieht wie folgt aus:

```
Imports System.ComponentModel
Imports BankingApplicationComponents

Public Class Account
    Implements INotifyPropertyChanged

    Public Event PropertyChanged As PropertyChangedEventHandler Implements INotifyPropertyChanged.PropertyChanged

    Protected Overridable Sub OnPropertyChanged(propname As String)
        RaiseEvent PropertyChanged(Me, New PropertyChangedEventArgs(propname))
    End Sub

    Private _accountNumber As String
    Private _acctName As String
    Private _bIC As String
    Private _iBAN As String
    Private _bACAccount As BACAccount

    Public Property AccountNumber As String
        Get
            Return _accountNumber
        End Get
        Set(value As String)
            _accountNumber = value
            OnPropertyChanged("AccountNumber")
        End Set
    End Property
    ... weitere Property-Eigenschaften

    Public Property BACAccount As BACAccount
        Get
```

```
            Return _bACAccount
        End Get
        Set(value As BACAccount)
            _bACAccount = value
        End Set
    End Property
End Class
```

Auflistung mit Kontakt-Elementen füllen

Wenn die Anwendung gestartet und somit das Fenster **MainWindow.xaml** geöffnet wird, sollen die Kontakte aus der Systemsteuerung eingelesen und in jeweils eine neue Klasse des Typs **Contact** geschrieben werden. Die neuen Instanzen wollen wir außerdem direkt zu einer Liste hinzufügen, die wir anschließend dem ersten **ComboBox**-Element zuweisen können. Wir nutzen dazu eine Funktion namens **GetBACContacts**, die eine Liste von Elementen des Typs **BACContact** liefert. Diese fügen wir einem Modul namens **DDBACObjects.vb** hinzu, dem wir außerdem noch per **Imports**-Anweisung die Elemente des Namespace **BankingApplicationComponents** hinzufügen.

Die folgende privat deklarierte Variable speichert die **Contacts**-Auflistung zwischen, damit diese nicht bei jedem Zugriff erneut ausgelesen werden müssen:

```
    Private m_Contacts As BACContacts
```

Die Funktion **GetBACContacts** liest die Kontakte mit den Methoden der DDBAC-Bibliothek aus der Systemsteuerung in die Variable **m_BACContacts** ein. Diese wird dann anschließend auch als Ergebnis der Funktion zurückgegeben. Wenn **m_BACContacts** bereits vorhanden ist, wird die Auflistung nicht erneut gefüllt. Einzige Ausnahme ist, wenn der Aufruf den Wert **True** für den Parameter **bolReset** enthält. In diesem Fall werden die Kontakte auf jeden Fall erneut eingelesen:

```
Public Function GetBACContacts(Optional bolReset As Boolean = False) As BACContacts
    If m_BACContacts Is Nothing Then
        m_BACContacts = New BACContacts
        m_BACContacts.Populate("")
    Else
        If bolReset = True Then
            m_BACContacts.Populate("")
        End If
    End If
    Return m_BACContacts
End Function
```

Damit haben wir mit **GetBACContacts** eine Liste der **BACContact**-Elemente eingelesen. Diese müssen wir nun noch in eine **ObservableCollection** mit Elementen des oben definierten Typs **Contact** umfüllen. Diese Liste wollen wir dann als Datenquelle des ersten **ComboBox**-Elements in unserem Fenster verwenden. Diese Aufgabe übernimmt die Funktion **GetContacts**. Sie

erstellt eine neue **ObservableCollection** in der Variablen **objContacts**. Dann durchläuft sie eine **For Each**-Schleife über alle **BACContact**-Elemente, welche die oben beschriebene Funktion **GetContacts** liefert. Innerhalb der Schleife legt sie ein neues **Contact**-Element an (nicht zu verwechseln mit **BACContact**), und schreibt die entsprechenden Werte in die Eigenschaften **BankCode**, **CountryCode** und **UserID** sowie einen Verweis auf das aktuelle **BACContact**-Objekt in die Eigenschaft **BACContact**. Anschließend fügt sie das neu erstellte und gefüllte Objekt zur ObservableCollection hinzu:

```
Public Function GetContacts() As ObservableCollection(Of Contact)
    Dim objContacts As ObservableCollection(Of Contact)
    Dim objContact As Contact
    Dim objBACContact As BACContact
    objContacts = New ObservableCollection(Of Contact)
    For Each objBACContact In GetBACContacts()
        objContact = New Contact
        With objContact
            .BankCode = objBACContact.BankCode
            .CountryCode = objBACContact.CountryCode
            .UserID = objBACContact.UserID
            .BACContact = objBACContact
            objContacts.Add(objContact)
        End With
    Next
    Return objContacts
End Function
```

Auflistung für die Konten füllen

Wenn der Benutzer in der gleich noch zu erstellenden ComboBox einen Kontakt ausgewählt hat, soll die zweite ComboBox die zu diesem Kontakt gehörenden Konten zur Auswahl anbieten. Deshalb benötigen wir analog zu den oben beschriebenen Funktionen **GetBACContacts** und **GetContacts** eine weitere Funktion.

Die Funktion **GetBACAccounts** erwartet als Parameter ein **BACContact**-Objekt, zu dem sie die **BACAccount**-Objekte einlesen soll. Sie verwendet dann schlicht die **Accounts**-Auflistung des **BACContact**-Objekts, um die Liste in der Variablen **m_Accounts** des Typs **BACAccounts** zu speichern:

```
Public Function GetBACAccounts(objBACContact As BACContact) As BACAccounts
    Return objBACContact.Accounts
End Function
```

Wenn der Benutzer im ersten **ComboBox**-Feld mit den Kontakten einen Eintrag ausgewählt hat, holen wir mit der folgenden Funktion die Daten zum Füllen des zweiten **ComboBox**-Feldes zur Anzeige der Konten dieses Kontakts. Dabei verwenden wir auch hier als Parameter das **BACContact**-Element des zu untersuchenden Kontaktes. Die Funktion erstellt als Erstes wieder eine neue ObservableCollection.

In einer **For Each**-Schleife durchlaufen wir diesmal die mit der Funktion **GetBACAccounts** geholten Konten. Innerhalb der **For Each**-Schleife legen wir ein neues Objekt des Typs **Account** an und weisen diesem die Werte für die Eigenschaften **AccountNumber**, **AcctName**, **IBAN** und **BIC** zu sowie einen Verweis auf das **BACAccount**-Objekt dieses Kontos. Anschließend fügen wir es der ObservableCollection namens **objAccounts** zu, die wir als Funktionsergebnis zurückliefern:

```
Public Function GetAccounts(objBACContact As BACContact) As ObservableCollection(Of Account)
    Dim objAccounts As ObservableCollection(Of Account)
    Dim objAccount As Account
    Dim objBACAccount As BACAccount
    objAccounts = New ObservableCollection(Of Account)
    For Each objBACAccount In GetBACAccounts(objBACContact)
        objAccount = New Account
        With objAccount
            .AccountNumber = objBACAccount.AccountNumber
            .AcctName = objBACAccount.AcctName
            .BIC = objBACAccount.BIC
            .IBAN = objBACAccount.IBAN
            .BACAccount = objBACAccount
        End With
        objAccounts.Add(objAccount)
    Next
    Return objAccounts
End Function
```

Fenster zur Anzeige der Auswahlfelder für Kontakte und Konten

Das Fenster soll im oberen Bereich zwei **ComboBox**-Elemente zur Auswahl von Kontakten und Konten anzeigen. Im Kontakt-Kombinationsfeld wollen wir die **UserID** und die **Bankleitzahl** anzeigen. Das Konto-Auswahlfeld soll nur die IBAN anzeigen. Die Definition dieser beiden Steuerelemente sieht wie folgt aus. Als Erstes das **ComboBox**-Element **cboContacts**, das an die Auflistung **Contacts** gebunden ist, die wir gleich noch im Code behind-Modul definieren.

Das **ComboBox**-Element hat außerdem ein Attribut für das Ereignis **SelectionChanged**. Da wir zwei Felder im **ComboBox**-Element anzeigen wollen, stellen wir diese in einem **ItemTemplate** und den entsprechenden untergeordneten Elementen wie folgt dar:

```
<Label Grid.Column="0" Content="Kontakt:"/>
<ComboBox x:Name="cboContacts" Grid.Row="0" Grid.Column="1" Width="200"
            ItemsSource="{Binding Contacts}"
            SelectedValuePath="UserID"
            SelectionChanged="CboContacts_SelectionChanged">
    <ComboBox.ItemTemplate>
        <DataTemplate>
            <TextBlock>
```

```
                <TextBlock.Text>
                    <MultiBinding StringFormat="{}{0} {1}">
                        <Binding Path="UserID" />
                        <Binding Path="BankCode" />
                    </MultiBinding>
                </TextBlock.Text>
            </TextBlock>
        </DataTemplate>
    </ComboBox.ItemTemplate>
</ComboBox>
```

Das zweite **ComboBox**-Element für die Konten ist an die Liste **Accounts** gebunden. Sie soll nur das Feld **IBAN** anzeigen und löst ebenfalls das Ereignis **SelectionChanged** aus:

```
<Label Grid.Column="0" Grid.Row="1" Content="Konto:"/>
<ComboBox x:Name="cboAccounts" Grid.Row="1" Grid.Column="1" Width="200"
            ItemsSource="{Binding Accounts}"
            SelectedValuePath="IBAN"
            SelectionChanged="CboAccounts_SelectionChanged"
            DisplayMemberPath="IBAN">
</ComboBox>
```

Code behind-Klasse des Fensters

Die Code behind-Klasse verwendet die folgenden beiden Namespaces:

```
Imports System.Collections.ObjectModel
Imports System.ComponentModel
```

AUSGABE 06/2020

DATENBANK ENTWICKLER

MAGAZIN FÜR DIE DATENBANKENTWICKLUNG MIT VISUAL STUDIO FÜR DESKTOP, WEB UND CO.

TOP-THEMEN:

STEUERELEMENTE	Das Kalendersteuerelement	SEITE 3
ANWENDUNGEN	Anwendungseinstellungen unter VB nutzen	SEITE 31
USER INTERFACE	Menüs und Symbolleisten mit WPF	SEITE 36
INTERAKTIV	XML-Dateien schreiben mit DOM	SEITE 55

André Minhorst Verlag

Das GridSplitter-Steuerelement

Das Grid-Steuerelement dürfte Ihnen als Leser dieses Magazins bekannt sein – wir verwenden es sehr oft, um die Position von Steuerelementen wie Bezeichnungsfeldern, Textfeldern oder Schaltflächen einzustellen. Im Zusammenhang mit dem Grid-Steuerelement müssen Sie ein weiteres Steuerelement unbedingt kennenlernen: das GridSplitter-Steuerelement. Dieses unterteilt das Grid zwischen zwei bestimmten Zeilen oder Spalten und bietet die Möglichkeit, dass Sie das Verhältnis der Höhe beziehungsweise Breite der Steuerelemente über/unter beziehungsweise links/rechts vom GridSplitter-Steuerelement einstellen können. Damit vergrößern Sie dann den jeweils benötigten Bereich.

Vielleicht kennen Sie ähnliche Lösungen von Access: Dort mussten Sie ein Rectangle-Steuerelement zwischen den zu vergrößernden/verkleinernden Steuerelementen platzieren. Mit dem Ereignis **Bei Mausbewegung** wurde dann beim Ziehen des Rechtecks zu einer Seite die Position des Rechtecks geändert beziehungsweise die Größe der Steuerelemente, die sich auf einer der beiden Seiten vom Rechteck befunden haben. Das Problem: Man musste die Größe und Position jedes Steuerelements, das durch die Größenänderung betroffen war, per Code einstellen.

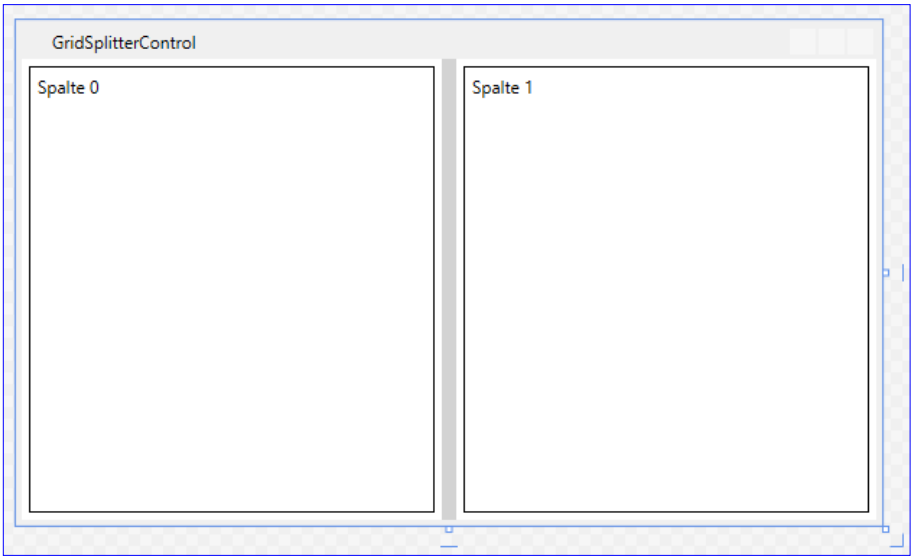


Bild 1: Das **GridSplitter**-Steuerelement zwischen zwei Steuerelementen

Vertikaler GridSplitter

Die Variante, die Sie mit dem **GridSplitter**-Steuerelement kennenlernen, ist ungleich komfortabler. Sie setzt allerdings voraus, dass Sie die Steuerelemente in den Spalten und Zeilen eines **Grid**-Steuerelements organisiert haben. Das **GridSplitter**-Steuerelement platzieren Sie dabei in einer eigens dafür vorgesehenen Spalte oder Zeile im Grid. Schauen wir uns das am Beispiel eines vertikalen GridSplitters an, der die Bereiche rechts und links vergrößern oder verkleinern soll – je nachdem, in welche Richtung Sie diesen bewegen. Der einfache Entwurf sieht wie in Bild 1 aus.

Den entsprechenden XAML-Code finden Sie hier: Wir haben mit den Elementen **Grid.ColumnDefinitions** und **ColumnDefinition** drei Spalten in einem Grid definiert. Die Spalten links und rechts erhalten für die Eigenschaft **Width** den Wert ***, was bedeutet, dass diese sich den Platz in der Breite aufteilen, der nicht von Spalten mit fest vorgegebener Breite eingenommen wird. Eine

solche Spalte legen wir in der Mitte an und weisen ihr die Breite **10** zu. Damit hat diese beim Öffnen die Breite 10 und die beiden äußeren Spalten teilen sich den restlichen Platz auf.

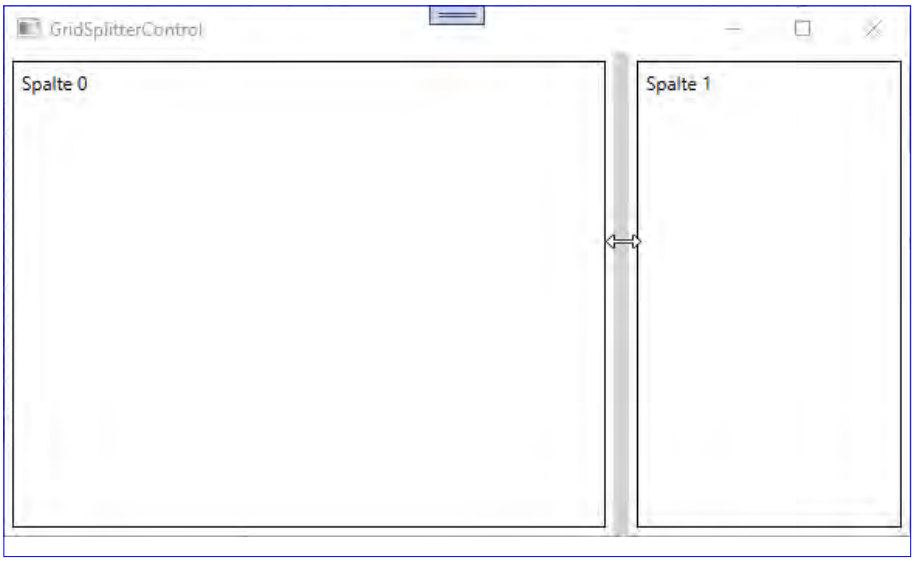


Bild 2: Einstellen der Größe der Bereiche links und rechts vom vertikalen **GridSplitter**-Steuerelement

Der linken und rechten Spalte fügen wir jeweils ein **Label**-Steuerelement zu, welche den kompletten Platz in den Spalten einnehmen. Der mittleren Spalte weisen wir ein **GridSplitter**-Steuerelement zu, das wir mit einem hellgrauen Hintergrund versehen und das durch den Wert **Stretch** für die Eigenschaft **HorizontalAlignment** die gesamte Breite der mittleren Spalte des Grids einnimmt. Außerdem legen wir vorsichtshalber den Wert des Attributs **ResizeDirection** auf **Columns** fest:

```
<Grid>
    <Grid.ColumnDefinitions>
        <ColumnDefinition Width="*"></ColumnDefinition>
        <ColumnDefinition Width="10"></ColumnDefinition>
        <ColumnDefinition Width="*"></ColumnDefinition>
    </Grid.ColumnDefinitions>
    <Label Grid.Column="0" BorderBrush="Black" BorderThickness="1">Spalte 0</Label>
    <GridSplitter Grid.Column="1" ResizeDirection="Columns" Background="LightGray" HorizontalAlignment="Stretch"></GridSplitter>
    <Label Grid.Column="2" BorderBrush="Black" BorderThickness="1">Spalte 1</Label>
</Grid>
```

Starten wir das Projekt, können wir den Gridsplitter per Herunterdrücken der linken Maustaste anfassen und durch Bewegen der Maus nach links oder rechts verschieben. Die Steuerelemente, die sich in der **Grid**-Spalte links und rechts davon befinden, werden dann entsprechend der Breite der jeweiligen Spalte angepasst (siehe Bild 2).

Horizontaler GridSplitter

Auf ähnliche Weise arbeiten wir mit einem **GridSplitter**-Steuerelement, mit dem wir das Grid in einen Teil über und einen Teil unter dem GridSplitter einteilen. Dazu stellen wir nun mehrere Zeilen her, von denen die mittlere das **GridSplitter**-Steuerelement aufnimmt:

```
<Grid>
    <Grid.RowDefinitions>
```

```
        <RowDefinition Height="*"></RowDefinition>
        <RowDefinition Height="10"></RowDefinition>
        <RowDefinition Height="*"></RowDefinition>
    </Grid.RowDefinitions>
    <Label Grid.Row="0" BorderBrush="Black" BorderThickness="1">Zeile 0</Label>
    <GridSplitter Grid.Row="1" ResizeDirection="Rows" Background="LightGray" HorizontalAlignment="Stretch"></GridSplitter>
    <Label Grid.Row="2" BorderBrush="Black" BorderThickness="1">Zeile 1</Label>
</Grid>
```

Wir brauchen an dem Beispiel gar nicht so viel zu ändern – wir legen statt des **Grid.ColumnDefinitions**-Elements ein **Grid.RowDefinitions**-Element an, ordnen diesen statt **ColumnDefinition**-Elementen **RowDefinition**-Elemente zu, die wir überdies statt mit der Eigenschaft **Width** mit der Eigenschaft **Height** ausstatten. Die oberste und die unterste Zeile erhalten den Wert ***** für die Höhe und die mittlere wieder den Wert **10**.

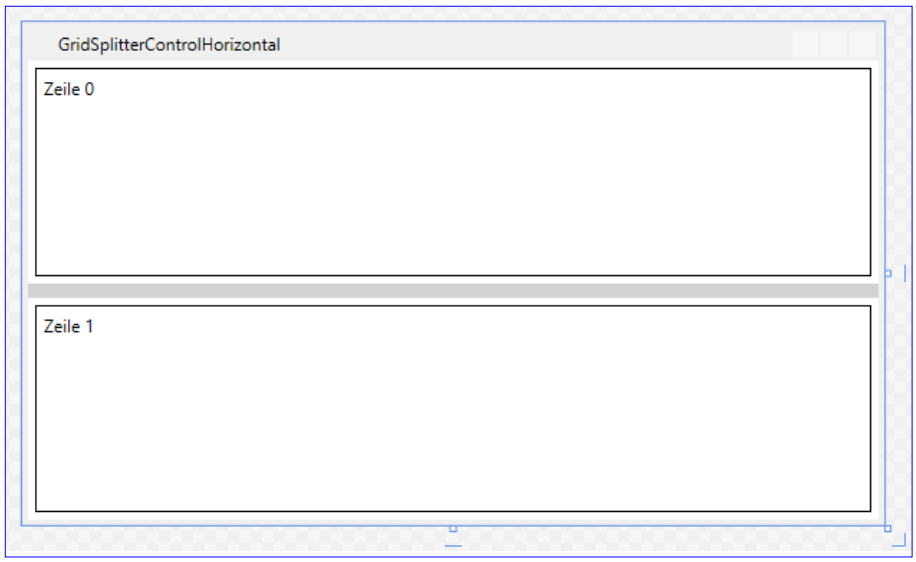


Bild 3: Entwurf des Grids mit dem horizontalen **GridSplitter**

Schließlich weisen wir die beiden Bezeichnungsfelder den Zeilen mit dem Index **0** beziehungsweise **2** und das **GridSplitter**-Element der Zeile mit dem Index **1** zu. Dieses statten wir außerdem mit dem Wert **Rows** für das Attribut **ResizeDirection** aus. Damit erhalten wir den Entwurf aus Bild 3.

Wenn wir die Anwendung starten, sieht das Ergebnis wie in Bild 4 aus.

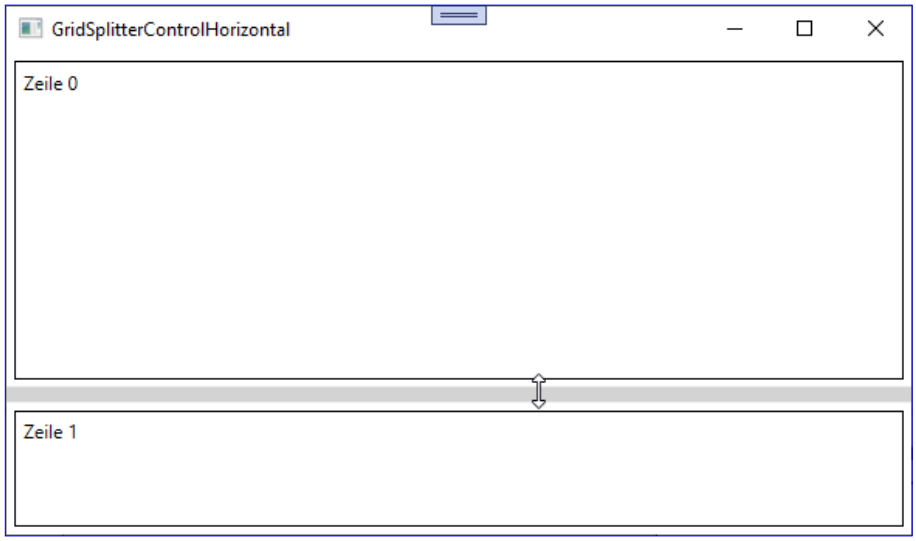


Bild 4: Einstellen der Höhe der Bereiche unter und über dem horizontalen **GridSplitter**

Das Attribut **ResizeDirection** ist standardmäßig auf den Wert **Auto** festgelegt, was in der Regel auch ausreicht – wir haben das Attribut hier nur auf die Werte **Columns** beziehungsweise **Rows** gesetzt, um seine Funktion zu dokumentieren.

GridSplitter mit Vorschau

In den bisherigen Beispielen wurden der GridSplitter und die dadurch beeinflussten Steuerelemente jeweils direkt durch das Ziehen mit der Maus angepasst.

Sie können auch festlegen, dass der Gridsplitter sich beim Ziehen mit der Maus bewegt, aber noch ohne seine Position oder die Größe der beteiligten Steuerelemente anzupassen (siehe Bild 5). Für diesen Fall stellen Sie das Attribut **ShowsPreview** auf den Wert **True** ein:

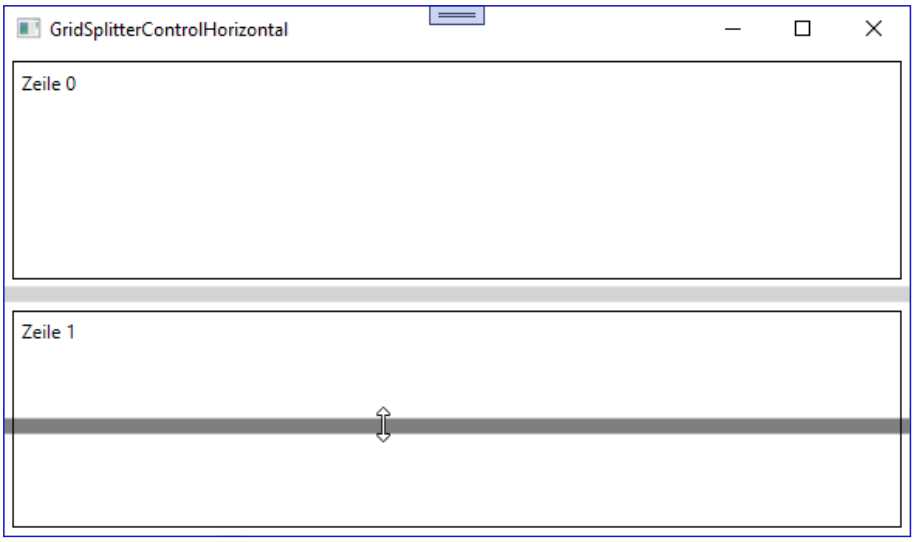


Bild 5: **GridSplitter**-Steuerelement mit Vorschau

```
<GridSplitter Grid.Row="1" ResizeDirection="Rows" Background="LightGray" HorizontalAlignment="Stretch"
ShowsPreview="True"></GridSplitter>
```

Mehrere GridSplitter

Sie können auch mehrere **GridSplitter**-Steuerelemente in einem Fenster nutzen. Im ersten Beispiel schauen wir uns an, wir Sie mehrere **GridSplitter**-Steuerelemente in einer Ebene nutzen können.

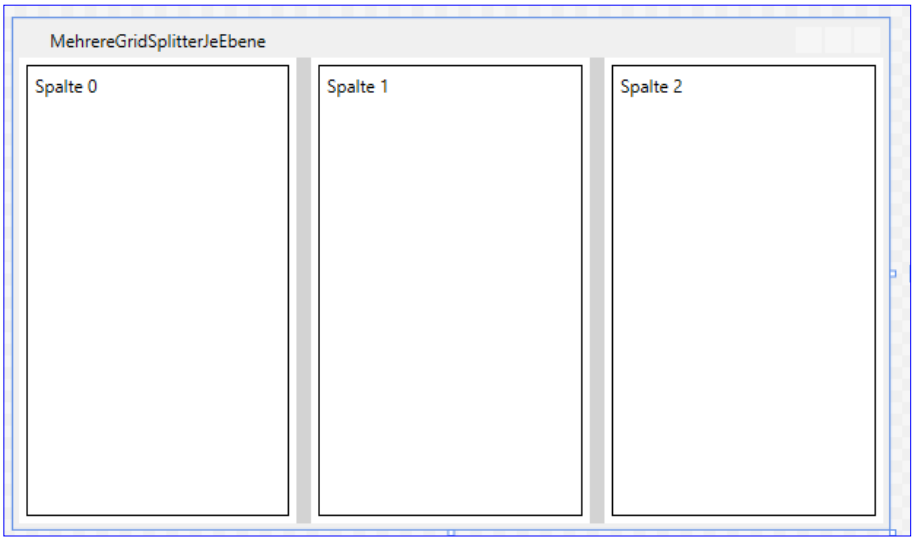


Bild 6: Mehrere **GridSplitter**-Steuerelement in einer Ebene, Entwurfsansicht

Dabei fügen wir zwei **ColumnDefinition**-Elemente zum Element **Grid.ColumnDefinitions** hinzu, von denen die letzte wieder * als Breite erhält und die vorletzte die Breite **10**. Außerdem legen wir für die letzte Spalte ein weiteres Label-Steuerelement an und für die vorletzte ein zweites **GridSplitter**-Steuerelement (Entwurf siehe Bild 6):

```
<Grid>
    <Grid.ColumnDefinitions>
        <ColumnDefinition Width="*"></ColumnDefinition>
        <ColumnDefinition Width="10"></ColumnDefinition>
        <ColumnDefinition Width="*"></ColumnDefinition>
```

Das Kalendersteuerelement

.NET bietet schon in der Grundausstattung sehr viele Steuerelemente für WPF-Anwendungen – hinzu kommen noch solche von Drittanbietern. Wir wollen nach und nach die WPF-Steuerelemente von .NET vorstellen, die wir noch nicht in den vorherigen Ausgaben beleuchtet haben. Und dort haben wir uns bisher ja nur um Standard-Steuerelemente wie TextBox, Button, ComboBox oder die diversen Steuerelemente zum Strukturieren anderer Steuerelemente gekümmert. Im vorliegenden Artikel schauen wir uns die Grundlagen zum Kalendersteuerelement an

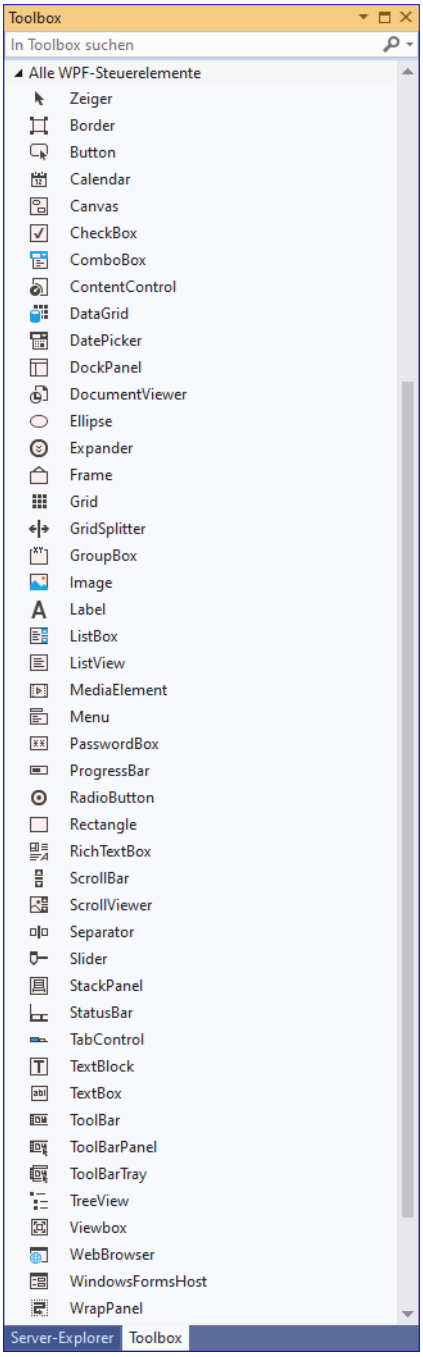


Bild 1: .WPF-Steuerelemente

Wenn Sie die Entwurfsansicht einer XAML-Datei anzeigen und die Toolbox aktivieren, sehen Sie im oberen Bereich namens **Häufig verwendete WPF-Steuerelemente** die oft verwendeten Steuerelemente wie **Button**, **CheckBox**, **ComboBox** oder **TextBox**. Der Bereich **Alle WPF-Steuerelemente** ist oft ausgeblendet, liefert aber noch viele weitere Steuerelemente, die Sie ohne weitere Handgriffe in Ihre Anwendungen integrieren können. Diesen Bereich sehen Sie in Bild 1.

Im vorliegenden Artikel wollen wir mit dem **Calendar**-Steuerelement beginnen. Dieses ziehen Sie im einfachsten Fall per Drag and Drop aus der Toolbox auf den aktuell geöffneten XAML-Entwurf (siehe Bild 2).

Im WPF-Code erscheint dann folgende Zeile:

```
<Calendar HorizontalAlignment="Left" Margin="33,27,0,0"
VerticalAlignment="Top"/>
```

In der laufenden Anwendung zeigt das **Calendar**-Steuerelement standardmäßig das aktuelle Datum an. Sie können andere Tage markieren, das aktuelle Datum bleibt jedoch farbig hinterlegt. Das **Calendar**-Steuerelement zeigt immer sechs Zeilen mit den Tagen jeweils einer Woche an. Da kein Monat 42 Tage enthält, werden die Tage des vorherigen und des folgenden Monats eingeblendet. Klicken Sie einen Tag an, der nicht zum aktuellen Monat gehört, wechselt die Ansicht zum Monat dieses Tages und markiert diesen

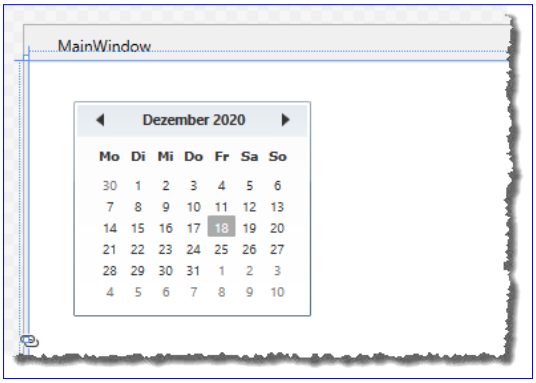


Bild 2: Das **Calendar**-Steuerelement

Tag. Eine andere Art, zum vorherigen oder zum nachfolgenden Monat zu wechseln, ist ein Mausklick auf einen der Pfeile links und rechts vom Monatsnamen.

Größe anpassen

Im Gegensatz zu vielen anderen Steuerelementen passt sich die Größe des **Calendar**-Steuerelements nicht automatisch an das umliegende Element an. Wenn Sie beispielsweise einem Grid ein **Button**-Element hinzufügen, nimmt dieses direkt den vollständigen Platz im Grid ein, solange Sie dies nicht mit entsprechenden Eigenschaften einschränken. Das Calendar-Steuerelement wird schlicht an der Stelle, an der es eingefügt wird, in der Standardgröße angezeigt.

Startdatum einstellen

Wenn Sie wollen, dass das **Calendar**-Steuerelement ein anderes Datum als das des heutigen Tages anzeigt, können Sie dies mit der Eigenschaft **DisplayDate** erreichen. Diese stellen Sie beispielsweise auf ein fixes Datum ein:

```
<Calendar DisplayDate="1.1.2021"></Calendar>
```

Bereits in der Entwurfsansicht zeigt das **Calendar**-Steuerelement dann den Monat **Januar 2021** an. Starten wir die Anwendung dann, bleibt es auch dabei – das **Calendar**-Steuerelement zeigt den **Januar 2021** an, aber markiert nicht den **1.1.2021**.

Liest man in der Dokumentation nach, erfährt man schnell, dass mit **DisplayDate** tatsächlich nur der Monat des mit **DisplayDate** angegebenen Datums eingestellt werden kann.

Wie aber können wir tatsächlich das Datum festlegen, das beim Öffnen markiert werden soll? Das erledigen Sie mit der Eigenschaft **SelectedDate**:

```
<Calendar SelectedDate="1.1.2021"></Calendar>
```

Damit wird das aktuelle Datum aber nach wie vor grau hinterlegt, während das mit **SelectedDate** angegebene Datum einen hellblauen Hintergrund erhält (siehe Bild 3).

Bild 3: Datum einstellen mit **SelectedDate**

Aktuelles Datum nicht markieren

Wenn Sie nicht möchten, dass das aktuelle Datum durch einen farbigen Hintergrund markiert wird, stellen Sie die Eigenschaft **IsTodayHighlighted** auf **False** ein:

```
<Calendar IsTodayHighlighted="False"></Calendar>
```

Anderen Wochentag als ersten Tag anzeigen

Normalerweise zeigt das **Calendar**-Steuerelement den Montag in der ersten Spalte an. Wenn Sie einen anderen Tag anzeigen wollen, beispielsweise den Samstag, legen Sie dies mit dem Attribut **FirstDayOfWeek** fest, das die englische Wochentagsbezeichnung erwartet – hier beispielsweise **Saturday**:

```
<Calendar FirstDayOfWeek="Saturday"></Calendar>
```

Einzelnes Datum auswählen

Das Attribut **SelectionMode** legt fest, ob einzelne Datumsangaben oder Datumsbereiche ausgewählt werden können. Dazu stellt diese Eigenschaft die folgenden Werte zur Verfügung:

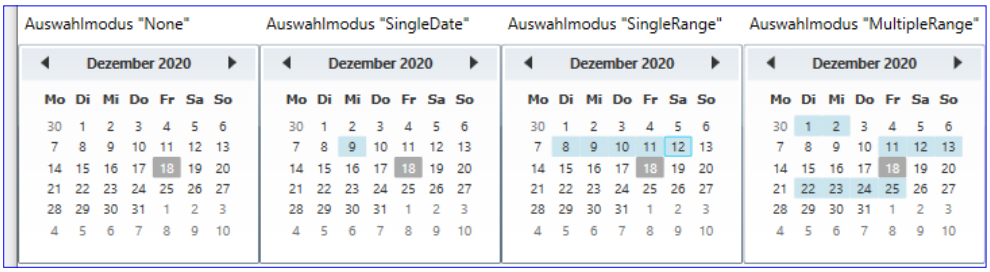


Bild 4: Verschiedene Auswahlmodi des **Calendar**-Steuerelements

- **None**: Es kann gar kein Datum ausgewählt werden.
- **SingleDate**: Es kann ein einzelnes Datum ausgewählt werden, und zwar per Mausklick auf das gewünschte Datum.
- **SingleRange**: Es kann ein einfacher Datumsbereich ausgewählt werden.
- **MultipleRange**: Es können mehrere Datumsbereiche ausgewählt werden.

Die Möglichkeiten dieser vier Einstellungen sehen Sie in Bild 4.

Einzelnen Datumsbereich auswählen

Wenn Sie die Einstellung **SingleRange** für die Eigenschaft **SelectionMode** angeben, können Sie einen oder mehrere zusammenhängende Tage auswählen. Das gelingt auf verschiedene Arten:

- Durch Anklicken des ersten Tages und Anklicken des letzten Tages bei gedrückter Umschalttaste.
- Durch Mausklick auf den ersten Tag und Ziehen der Markierung bei gedrückter Maustaste bis zum letzten Tag.

Wenn Sie einen Zeitraum wählen wollen, der sich über den aktuell angezeigten Monat hinaus erstreckt, können Sie den ersten Tag markieren, dann mit den **Nach links**- oder **Nach rechts**-Schaltflächen zum gewünschten Monat wechseln und dann bei gedrückter Umschalttaste den letzten Tag des Datumsbereichs markieren.

Oder Sie markieren den ersten Tag und klicken dann bei gedrückter Umschalttaste jeweils auf einen der Tage, die zwar im Kalender angezeigt werden, aber nicht zum aktuellen Monat gehören und wechseln so zum vorherigen oder folgenden Monat. Haben Sie den Zielmonat erreicht, klicken Sie – immer noch bei gedrückter Umschalttaste – auf den gewünschten Tag.

Mehrere Datumsbereiche auswählen

Wählen Sie die Einstellung **MultipleRange** für **SelectionMode** aus, können Sie sogar mehrere Bereiche markieren. Für den ersten Bereich klicken Sie auf das erste Startdatum und ziehen dann die Maus bei gedrückter Maustaste bis zum Ende des ersten Bereichs. Für die folgenden Bereiche drücken Sie die **Strg**-Taste und fügen die weiteren Bereiche auf die gleiche Weise

Commands verwenden

Wenn Sie unter Access und VBA Ereignisse programmiert haben, hat beispielsweise jedes Steuerelement, jede Ribbon-Schaltfläche und jeder Kontextmenü-Eintrag ein eigenes Ereignis erhalten – auch, wenn diese genau die gleiche Funktion auslösen sollten. Das war schon durch die unterschiedlichen Schnittstellen nötig. Unter WPF können Sie ein und dieselbe Funktion von verschiedenen Steuerelementen aus auslösen. Das gelingt allerdings nicht mit herkömmlichen Ereignismethoden, sondern mit sogenannten Commands. Was das ist und wie Sie es in Ihre Anwendungen einbauen können, zeigt der vorliegende Artikel.

Vorteil: Alles an einem Ort

Neben der Möglichkeit, benutzerdefinierte Ereignisse durch Commands abzubilden und diese von überall in der Anwendung aufzurufen, bieten Commands noch weitere Möglichkeiten: So gibt es auch eingebaute Commands, die etwa Basisfunktionalitäten der Zwischenablage wie Kopieren, Ausschneiden oder Einfügen bereitstellen. Wenn Sie einmal mit Access programmiert haben, wissen Sie, dass diese Funktionen immer in Form von API-Funktionen selbst bereitgestellt werden mussten und Sie nicht einfach die Systemfunktionen nutzen konnten. Diese würden Sie dann von jedem Ort, wo diese Funktion benötigt wird, aufrufen müssen, und zwar von der dafür vorgesehenen Ereignisporzedur aus. Wir schauen uns also in diesem Artikel zwei verschiedene Typen von Commands an: benutzerdefinierte und eingebaute Commands.

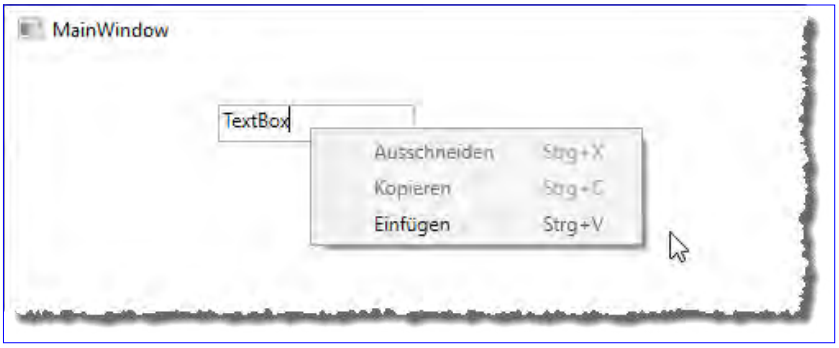


Bild 1: Aktivierte und deaktivierte Kontextmenü-Einträge

Vorteil: Tastenkombinationen

Ein weiterer Vorteil von Commands ist, dass Sie für ein benutzerdefiniertes Command eine Tastenkombination definieren können beziehungsweise, dass die Tastenkombinationen für eingebaute Commands bereits vorhanden sind – wie beispielsweise **Strg + C**, **Strg + X** oder **Strg + V** für **Kopieren**, **Ausschneiden** und **Einfügen**.

Vorteil: Aktivieren und Deaktivieren

Commands wie die zum Kopieren, Ausschneiden oder Einfügen des aktuell markierten Inhalts werden automatisch aktiviert oder deaktiviert – je nachdem, ob gerade ein Inhalt zum Kopieren oder Ausschneiden markiert beziehungsweise ob die Zwischenablage überhaupt Inhalt enthält, der aktuell eingefügt werden kann (siehe Bild 1). Auch für benutzerdefinierte Commands können Sie an einer Stelle festlegen, ob und wann diese aktiviert oder deaktiviert sind – abhängig vom jeweiligen Kontext. Während das für die Funktionen der Zwischenablage automatisch funktioniert, schauen wir uns später auch an, wie Sie benutzerdefinierte Commands von einer Stelle aus aktivieren und deaktivieren.

Definition von Commands

Wenn Sie ein Command definieren, erledigen Sie das unter Verwendung der **ICommand**-Schnittstelle. Diese sieht vor, dass Sie die folgenden beiden Methoden implementieren:

- **Execute**: Enthält die Anweisungen, die beim Aufruf des Commands ausgeführt werden sollen.
- **CanExecute**: Enthält Code, der festlegt, ob das Command gerade aktiviert oder deaktiviert ist.

Commands aufrufen

Um ein solches Command aufzurufen, müssen Sie für das jeweilige Steuerelement über ein sogenanntes Command binding festlegen, welches Command für das jeweilige Ereignis aufgerufen werden soll. Für eingebaute Commands ist dies einfacher als für benutzerdefinierte Commands. Wir schauen uns das für ein eingebautes Command an einem einfachen Beispiel an.

Dabei wollen wir die drei Befehle zum Ausschneiden, Kopieren und Einfügen in drei Schaltflächen unterbringen und mit diesen den Inhalt eines Textfeldes bearbeiten können. Der WPF-Code dafür sieht wie folgt aus:

```
<StackPanel Orientation="Horizontal">
    <Button x:Name="cmdAusschneiden" Command="ApplicationCommands.Cut">Ausschneiden</Button>
    <Button x:Name="cmdKopieren" Command="ApplicationCommands.Copy">Kopieren</Button>
    <Button x:Name="cmdEinfuegen" Command="ApplicationCommands.Paste">Einfügen</Button>
</StackPanel>
<TextBox x:Name="txtText" Grid.Row="1" Width="200"/>
```

Wir haben hier herkömmliche Schaltflächen angelegt, für die wir allerdings nicht das **Click**-Ereignis definiert haben, sondern mit dem **Command**-Attribut ein Command definiert haben, das beim Anklicken der Schaltfläche ausgelöst werden soll.

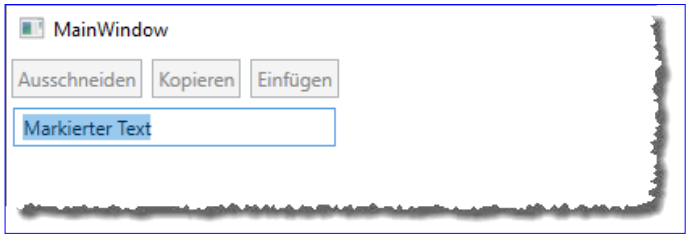


Bild 2: Schaltflächen, die nicht aktiviert werden

Das Ergebnis sieht wie in Bild 2 aus. Leider werden die Schaltflächen nicht aktiviert oder deaktiviert, wenn man einen Teil des Textes im Textfeld markiert. Dazu fehlt freilich noch ein Attribut – wir haben nämlich noch nicht festgelegt, auf welches Element der Benutzeroberfläche sich die Commands beziehen sollen. Das legen wir mit dem Attribut **CommandTarget** fest, welches per Binding das entsprechende Steuerelement referenziert:

```
<Button x:Name="cmdAusschneiden" Command="ApplicationCommands.Cut"
    CommandTarget="{Binding ElementName=txtText}">Ausschneiden</Button>
<Button x:Name="cmdKopieren" Command="ApplicationCommands.Copy"
    CommandTarget="{Binding ElementName=txtText}">Kopieren</Button>
<Button x:Name="cmdEinfuegen" Command="ApplicationCommands.Paste"
    CommandTarget="{Binding ElementName=txtText}">Einfügen</Button>
```

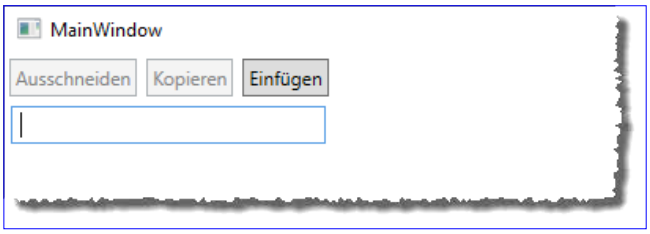


Bild 3: Aktivierte **Einfügen**-Schaltfläche

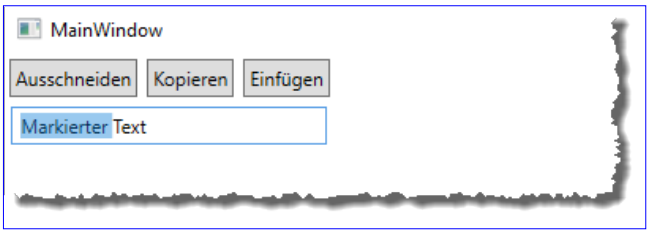


Bild 4: Durch Markierung aktivierte **Ausschneiden**- und **Einfügen**-Schaltflächen

Wenn das Textfeld leer ist, die Zwischenablage jedoch einen Wert enthält, der in das Textfeld eingefügt werden kann, wird nur die **Einfügen**-Schaltfläche aktiviert – und sie fügt den Inhalt der Zwischenablage auch per Mausklick ein (siehe Bild 3). Wenn sich beispielsweise ein Objekt wie eine Datei in der Zwischenablage befindet, wird die Einfügen-Schaltfläche nicht aktiviert, denn ein solches Objekt kann nicht in das Textfeld eingefügt werden. Gleiches gilt, wenn die Zwischenablage komplett leer ist.

Hat der Benutzer zusätzlich noch mindestens ein Zeichen im Textfeld **txtText** markiert, werden auch die beiden Schaltflächen zum Ausschneiden und Kopieren aktiviert (siehe Bild 4). Und auch das Ausschneiden und Kopieren mit den Schaltflächen funktioniert ohne Probleme.

Hier geschehen also einige Dinge automatisch – sowohl die Aktionen, die durch die Schaltflächen ausgeführt werden sollen, werden durch die Zuweisung des entsprechenden Wertes zum Attribut **Command** automatisch ausgeführt als auch die Aktivierung oder Deaktivierung abhängig vom Status des mit **CommandTarget** angegebenen Steuerelements.

Commands mit IntelliSense

Weiter oben haben wir einfach das Command als Wert der **Command**-Eigenschaft angegeben. Wenn Sie das auch gemacht haben, ist Ihnen vielleicht aufgefallen, dass es hier keine IntelliSense-Unterstützung gab, was für Visual Studio eher ungewöhnlich ist. Mit einer etwas anderen Schreibweise gelangen Sie allerdings schnell zur Anzeige der möglichen Befehle per IntelliSense. Diese sieht wie folgt aus – hier am Beispiel der Ausschneiden-Schaltfläche:

```
<Button x:Name="cmdAusschneiden" Command="{x:Static ApplicationCommands.Cut}"
    CommandTarget="{Binding ElementName=txtText}">Ausschneiden</Button>
```

Eingebaute Commands überschreiben

Wenn Sie schon eingebaute Commands nutzen können, um von verschiedenen Steuerelementen darauf zugreifen zu können, dann wäre es doch praktisch, wenn man diese eingebauten Commands auch überschreiben könnte. Damit meinen wir beispielsweise, dass Sie die Ausschneiden-Aktion **ApplicationCommands.Cut** durch eine eigene Methode ersetzen, die vor dem Ausschneiden noch nachfragt, ob der Inhalt wirklich ausgeschnitten werden soll.

Dazu können Sie mit dem Executed-Attribut eine statt der eigentlichen Funktion einzusetzende Methode angeben und diese dann im Code behind-Modul des WPF-Fensters implementieren. Dazu fügen wir in der XAML-Datei ein Element namens **TextBox.CommandBindings** mit dem Unterelement **CommandBinding** hinzu, das die drei Attribute **Command**, **CanExecute** und **Executed** enthält – und zwar für das Textfeld, für das wir das Command anpassen wollen:

```
<TextBox x:Name="txtText" Grid.Row="1" Width="200">
    <TextBox.CommandBindings>
        <CommandBinding Command="{x:Static ApplicationCommands.Cut}" CanExecute="CommandBinding_CanExecute"
            Executed="CommandBinding_Executed" />
    </TextBox.CommandBindings>
</TextBox>
```

Hier tragen wir ein, dass es sich um das **Command**-Element **ApplicationCommands.Cut** handelt und wie die auszuführenden Methoden für **CanExecute** und **Executed** heißen.

Den Rest der Definition für die Schaltfläche **cmdAusschneiden** behalten wir bei:

```
<Grid>
    <StackPanel Orientation="Horizontal">
            <Button x:Name="cmdAusschneiden" Command="{x:Static ApplicationCommands.Cut}"
                CommandTarget="{Binding ElementName=txtText}" Content="Ausschneiden" >
            </Button>
        ...
</Grid>
```

In der Code behind-Klasse fügen wir die beiden für die Attribute **CanExecute** und **Executed** angegebenen Methoden hinzu. Um das möglichst einfach zu gestalten, erledigen wir das wie folgt:

- Tippen Sie den Namen des Attributs ein, also beispielsweise **Executed**.
- Geben Sie das Gleichheitszeichen ein. Es erscheint ein Popup mit zwei Befehlen, von denen Sie den ersten namens **<Neuer Ereignishandler>** wählen.
- Das legt automatisch die Ereignismethode mit den passenden Parametern in der Code behind-Datei an.

Die Code behind-Klasse sieht nach dem Anlegen der beiden Ereignismethoden **CommandBinding_CanExecute** und **CommandBinding_Executed** wie folgt aus:

```
Class MainWindow
    Private Sub CommandBinding_CanExecute(sender As Object, e As CanExecuteRoutedEventArgs)
        e.CanExecute = True
    End Sub

    Private Sub CommandBinding_Executed(sender As Object, e As ExecutedRoutedEventArgs)
        MessageBox.Show("Überschriebenes Ausschneiden-Command")
    End Sub
```

```
End Class
```

Die Methode **CanExecute** liefert für dieses Beispiel immer den Wert **True** zurück. Die Methode **CommandBinding_Executed** zeigt eine Meldung an, wenn der Benutzer dieses Command aufruft.

Sie könnten hier beispielsweise abfragen, ob der Inhalt wirklich ausgeschnitten werden soll und, falls der Benutzer dem zustimmt, den Inhalt ausschneiden. Dazu fällt uns zunächst die folgende Möglichkeit ein – wir verwenden einfach das **Command**-Element, das wir mit der **Execute**-Methode auch per Code aufrufen können:

```
    Private Sub CommandBinding_Executed(sender As Object, e As ExecutedRoutedEventArgs)
        MessageBox.Show("Überschriebenes Ausschneiden-Command")
        ApplicationCommands.Cut.Execute(txtText, txtText)
    End Sub
```

Der **Execute**-Methode übergeben wir einen Verweis auf das **TextBox**-Steuerelement, damit diese weiß, welchen Inhalt sie ausschneiden soll. Damit handeln wir uns allerdings eine Endlosschleife ein, aus der wir nur herauskommen, indem wir zu Visual Studio wechseln und das Debuggen dort mit der Tastenkombination **Umschalt + F5** beenden. Logisch: Wenn wir das **Command**-Element **ApplicationCommands.Cut** aufrufen, wird dieses natürlich wie definiert erneut aufgerufen – und löst damit auch immer wieder das **Executed**-Ereignis aus.

Zum Glück gibt es noch weitere Möglichkeiten, den Inhalt eines Textfeldes auszuschneiden. Wie man die Zwischenablage mit VB programmiert, zeigen wir im Artikel **Zwischenablage programmieren** (**www.datenbankentwickler.net/218**). Dort erfahren wir auch, dass das **TextBox**-Steuerelement die Methoden **Cut, Copy** und **Paste** bereitstellt, um den markierten Inhalt auszuschneiden, zu kopieren oder mit dem Inhalt der Zwischenablage zu überschreiben.

Wir brauchen die Methode also nur wie folgt zu ändern:

```
Private Sub CommandBinding_Executed(sender As Object, e As ExecutedRoutedEventArgs)
    MessageBox.Show("Überschriebenes Ausschneiden-Command")
    e.Handled = True
    txtText.Cut()
End Sub
```

Danach ruft sich diese nicht mehr endlos auf, weil wir so nicht immer wieder das **Ausschneiden**-Command auslösen.

Schaltfläche nur aktivieren, wenn Markierung vorliegt

Zuvor haben wir die Eigenschaft **CanExecute** in der Methode **CommandBinding_CanExecute** standardmäßig auf den Wert **True** eingestellt. Nun wollen wir diese Methode nutzen, um per Code zu ermitteln, ob eine Markierung vorliegt und nur dann die Schaltfläche **cmdAusschneiden** aktivieren. Dazu fragen wir in dieser Methode ab, ob die Eigenschaft **Length** von **txtText.SelectedText** den Wert **True** zurückliefert. Nur dann soll mit **e.CanExecute** und dem Wert **True** festgelegt werden, dass der **Ausschneiden**-Befehl und somit die Schaltfläche aktiviert ist. Anderenfalls stellen wir **e.CanExecute** auf den Wert **False** ein:

Anwendungseinstellungen unter VB nutzen

Es gibt verschiedene Gründe, Konfigurationsdaten einer Anwendung zu speichern – um den zuletzt verwendeten Dateipfad zu speichern, die Position und Größe von Fenstern zu sichern oder auch um die Verbindungsdaten zu einer Datenbank, die sich gelegentlich ändern, zu hinterlegen. Es gibt dafür einen eigenen Bereich in einem Projekt, der in einer speziellen Datei gespeichert wird. Dieser ist auch per VB zugänglich. Wie Sie Konfigurationsdateien mit Visual Basic anlegen und diese wieder abrufen, zeigt der vorliegende Artikel.

Es gibt einige mögliche Orte zum Speichern von Konfigurationsdaten – eine Textdatei, eine XML-Datei, eine Datenbank oder auch die vom Projekt bereitgestellte Möglichkeit der Konfigurationsdateien. Diese wollen wir in diesem Artikel vorstellen, denn sie ist die einfachste Variante. Diese finden Sie, wenn Sie im Projektmappen-Explorer doppelt auf den Eintrag **My Project** klicken. Es erscheint dann der Dialog aus Bild 1, in dem Sie auf die Seite **Einstellungen** wechseln.

Hier finden Sie eine Liste, in die Sie beliebig viele Einträge schreiben können. Jeder Eintrag der Liste enthält vier Informationen:

- **Name**: Bezeichnung der Einstellung
- **Typ**: Datentyp der Einstellung. Abhängig von der Auswahl finden Sie anschließend verschiedene Möglichkeiten im Feld **Wert**, um den jeweiligen Wert einzufügen. Für den Wert **Boolean** können Sie **True** oder **False** aus einem Auswahlfeld wählen, für

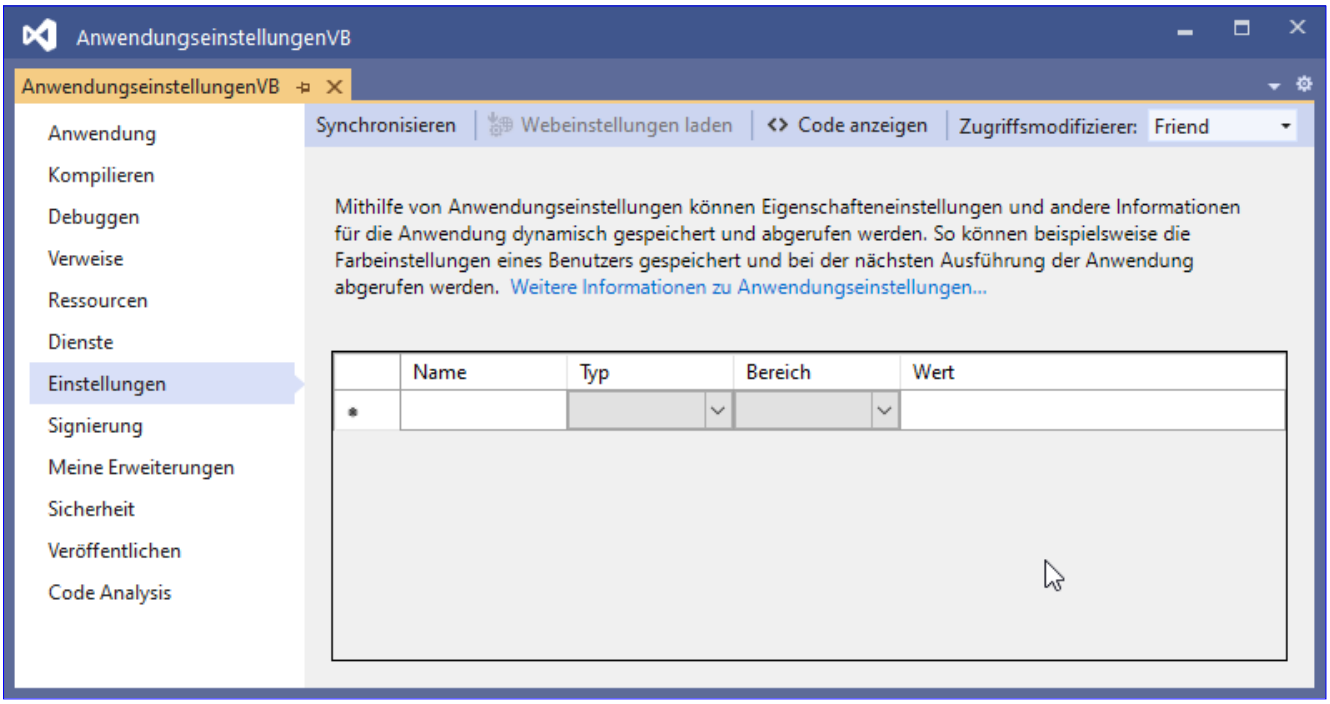


Bild 1: Der Bereich zum Verwalten von Anwendungseinstellungen

Date wird ein DatePicker angezeigt, bei Farben ein Farbdialog und so weiter (siehe Bild 2).

- **Bereich**: Gültigkeitsbereich mit den möglichen Werten **Anwendung** oder **Benutzer**.

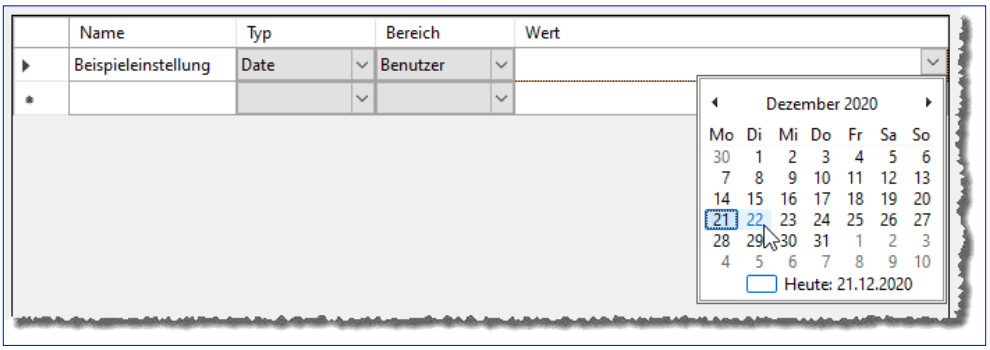


Bild 2: Verschiedene Eingabemöglichkeiten je nach Datentyp

- **Wert**: Wert der Einstellung

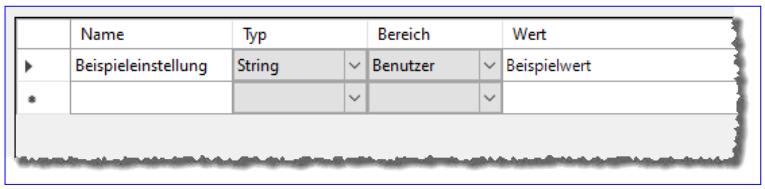


Bild 3: Beispiel für Anwendungseinstellungen

Nun wollen wir diesen Bereich nur initial nutzen, um die beim ersten Start einer Anwendung benötigten Voreinstellungen zu hinterlegen. Wir legen als Beispiel einfach eine Einstellung namens **Beispieleinstellung** mit dem Datentyp **String** und dem Wert **Beispielwert** an. Diese Werte sind einfach festzulegen (siehe Bild 3).

Bereich: Anwendung oder Benutzer?

Einen genaueren Blick werfen wir auf die Spalte **Bereich**. Hier können Sie zwischen den folgenden Werten wählen:

- **Anwendung**: Diese Einstellungen betreffen die Anwendung im Allgemeinen. Sie können nicht durch den Benutzer geändert werden.

- **Benutzer**: Diese Einstellungen sind benutzerspezifisch – zum Beispiel zum Speichern von Verzeichnissen oder Größe und Position von Fenstern. Diese Werte können dementsprechend vom Benutzer geändert werden.

Da wir zeigen wollen, wie wir die Einstellungen lesen und schreiben, legen wir für unsere erste Beispieleinstellung den Bereich **Benutzer** fest.

Speicherort der Einstellungen zur Design-Zeit

Bevor wir den Zugriff auf die Einstellungen per Code zeigen, schauen wir uns noch den Speicherort der Anwendungseinstellungen an. Dieser ist die Datei **app.config** im Projektverzeichnis. Die Datei sieht wie in Bild 5 aus und speichert die Anwendungseinstellungen für den Benutzer im Unterelement **userSettings**.

Dort finden Sie im Unterelement **<AnwendungseinstellungenVB.MySettings>** (wobei **AnwendungseinstellungenVB** der Projektname ist) für jede Einstellung einen Eintrag namens **setting**. **setting** speichert die im Anwendungseinstellungen-Dialog abgefragten ersten beiden Spalten in den Attributen **name** und **serializeAs** (Typ). Der Bereich ist ja bereits durch **userSettings** gegeben. Der Wert wird im Unterelement **value** gespeichert. Die Anwendungseinstellungen auf Anwendungsebene werden im Unterelement **applicationSettings** gespeichert.

Auf die Einstellung zugreifen

Wir testen die Einstellungen und den Zugriff darauf in einem WPF/VB-Projekt. Im Fenster **MainWindow.xaml** legen wir ein **TextBox**-Steuerelement an, das direkt nach dem Start den Wert der Einstellung anzeigen soll. Wie greifen wir auf die Einstellungen zu? Wir verwenden einen Namespace namens **My**, der über verschiedene Klassen Zugriff auf Informationen des aktuellen Projekts bietet:

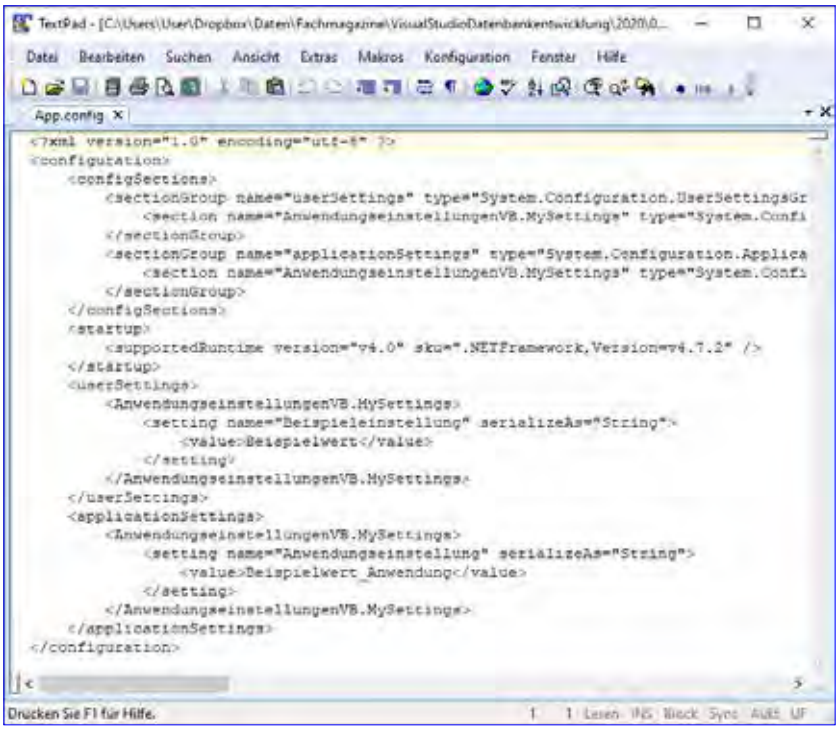


Bild 5: Datei mit unserer Beispiel-Anwendungseinstellung

- **My.Application**: Liefert Informationen über die aktuelle Instanz der Anwendung.
- **My.Computer**: Erleichtert den Zugriff auf Computer-bezogene Elemente wie die Zwischenablage, die Registry und so weiter.
- **My.Settings**: Der Bereich, den wir im vorliegenden Artikel untersuchen und der den Zugriff auf die Anwendungseinstellungen regelt.
- **My.Resources**: Zugriff auf die Ressourcen des aktuellen Projekts.
- **My.User**: Liefert Informationen über den aktuellen Benutzer.

Die Klasse **My.Settings** bietet im VB-Editor über IntelliSense nicht nur die eingebauten Eigenschaften, Methoden und Ereignisse an, sondern wir finden hier auch schon einen Eintrag für die von uns auf der Seite **Einstellungen** angelegte Beispieleinstellung (siehe Bild 4).

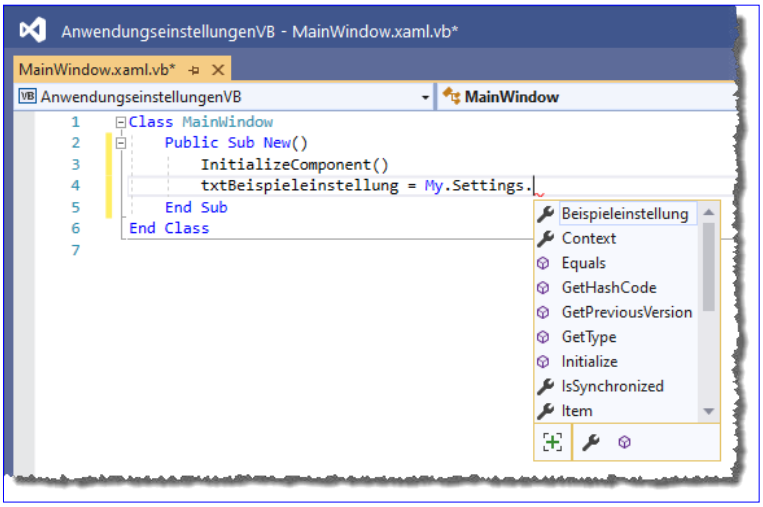


Bild 4: Zugriff auf eine Anwendungseinstellung

Im folgenden schauen wir uns diese Eigenschaften, Methoden und Ereignisse an:

- **<Name der Einstellung>**: Liefert den Wert der mit diesem Namen versehenen Einstellung.

Das Menu-Steuerelement

Wie Sie einer Anwendung beziehungsweise einem Fenster einer WPF-Anwendung ein Ribbon hinzufügen können, haben wir an anderer Stelle bereits erläutert. Wer nicht auf die eher platzraubenden Ribbons steht, kann sich auch der klassischen Menüs bedienen. Diese lassen sich unter WPF genauso leicht definieren wie Ribbons. Dieser Artikel zeigt, wie Sie einem Fenster ein Menü hinzufügen und welche Steuerelemente sich dort unterbringen lassen. Damit die Optik nicht leidet, bauen wir auch hier passende Icons ein.

Grundstruktur des Menüs

Normalerweise enthält ein **Window**-Element immer ein **Grid**-Element zur Strukturierung der enthaltenen Steuerelemente. Wenn Sie ein Menü hinzufügen wollen, strukturieren wir den XAML-Code ein wenig um. Als oberstes Element unterhalb des **Window**-Elements verwenden wir dann nämlich ein **DockPanel**-Element. Wenn wir diesem ein **Menu**-Element und ein **Grid**-Element hinzufügen, werden beide untereinander angeordnet und Sie können im **Grid**-Element wie üblich die weiteren Steuerelemente anordnen. Im folgenden Beispiel fügen wir erst einmal ein ganz einfaches Menü hinzu, das Sie im Entwurf in Bild 1 sehen. Das **DockPanel**-Element enthält zuerst ein **Menu**-Element, das Hauptelement eines Menüs. Diesem fügen Sie beliebige **MenuItem**-Elemente zur obersten Ebene hinzu, die alle oben in der Menüleiste erscheinen. Fügen Sie den **MenuItem**-Elementen der obersten Ebene weitere **MenuItem**-Elemente hinzu, werden diese darunter angeordnet. Zur Laufzeit erscheinen diese nur, wenn der Benutzer den **MenuItem**-Eintrag der obersten Ebene öffnet. Die Beschriftung der Menüeinträge stellen wir mit dem **Header**-Attribut ein:

```
<Window x:Class="MainWindow" ... Title="MainWindow" Height="450" Width="800">
    <DockPanel>
        <Menu DockPanel.Dock="Top">
            <MenuItem Header="Datei">
                <MenuItem Header="Beenden"></MenuItem>
            </MenuItem>
        </Menu>
        <Grid>

        </Grid>
    </DockPanel>
</Window>
```

Starten wir die Anwendung, erscheint das Menü wie in Bild 2.

Bild 1: Einfaches Menü

Menüeinträge per Alt-Tastenkombination steuern

Von Windows-Anwendungen, die noch mit dem herkömmlichen Menüsystem arbeiten, kennen Sie sicher die Möglichkeit, ein Menü aufzuklappen oder einen Menüpunkt auszuführen, indem Sie eine bestimmte Tastenkombination betätigen. Diese besteht

aus der Taste Alt und dem Buchstaben der Bezeichnung des gewünschten Menüeintrags, der unterstrichen dargestellt wird.

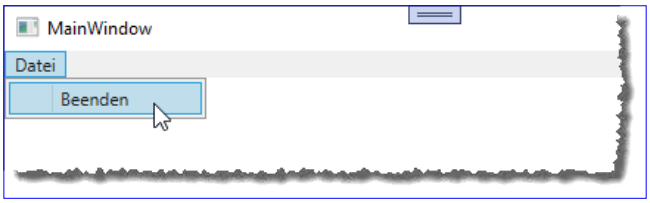


Bild 2: Einfaches Menü zur Laufzeit

Solche Menüeinträge mit unterstrichenen Buchstaben lassen sich auch für die WPF-Menüs realisieren. Dazu fügen Sie nur vor dem Buchstaben, der für die Tastenkombination genutzt werden soll, einen Unterstrich ein wie in den folgenden Beispieleinträgen:

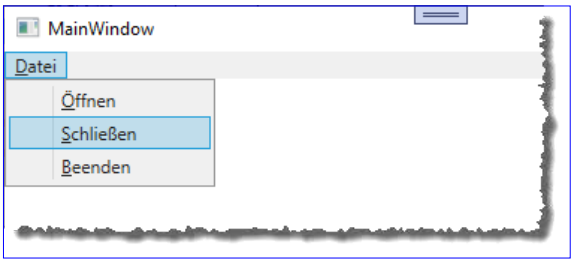


Bild 3: Menü mit Tastenkombinationen

```
<Menu DockPanel.Dock="Top">
    <MenuItem Header="_Datei">
        <MenuItem Header="_Öffnen"></MenuItem>
        <MenuItem Header="_Schließen"></MenuItem>
        <MenuItem Header="_Beenden"></MenuItem>
    </MenuItem>
</Menu>
```

Die Hauptmenüpunkte sehen danach zunächst wie üblich aus. Wenn Sie jedoch die **Alt**-Taste herunterdrücken und gedrückt halten, werden die Buchstaben mit dem führenden Unterstrich unterstrichen dargestellt. Betätigen Sie also in diesem Fall bei gedrückter **Alt**-Taste noch den Buchstaben **D**, wird das Untermenü wie in Bild 3 geöffnet. Dann können Sie mit dem entsprechenden weiteren Buchstaben einen der Untermenüpunkte auswählen. Drücken Sie im initialen Zustand nur die **Alt**-Taste und lassen diese wieder los, wird der erste Menüpunkt blau markiert und die mit dem Unterstrich versehenen Buchstaben in den Bezeichnungen werden ebenfalls nun unterstrichen dargestellt. Von nun an können Sie mit der Taste **Nach unten** die Untermenüpunkte anzeigen und dann mit den Tasten **Nach unten** und **Nach oben** einen der Menüpunkte auswählen und dann mit der Eingabetaste aufrufen. Wenn das Hauptmenü bereits mehrere Einträge hätte, könnten Sie nach dem Betätigen der **Alt**-Taste mit den Tasten **Nach links** und **Nach rechts** zwischen den Menüpunkten des Hauptmenüs navigieren.

Icons zu den Menüeinträgen hinzufügen

Ein Menü ist nur halb so hübsch, wenn es keine Icons aufweist. Also fügen wir dem Menü ein paar Icons hinzu. Diese müssen Sie zunächst zum Projekt hinzufügen. Dazu legen wir im Projektmappen-Explorer ein neues Verzeichnis an. Diesem fügen wir dann die benötigten Dateien hinzu.

Die Icons lassen sich nicht so eben per Attribut zum Menüeintrag hinzufügen. Dazu legen wir zunächst ein Unterelement namens **MenuItem.Icon** an. Dieses enthält ein weiteres Unterelement namens **Image**, für dessen **Source**-Attribut wir schließlich den Verweis auf das zu verwendende Bild angeben. Dazu geben Sie den Pfad zu der Datei an, wie er im Projektmappen-Explorer vorliegt, also zum Beispiel **/Icons/add.png**:

```
<MenuItem Header="Einträge mit Icons">
    <MenuItem Header="_Hinzufügen">
        <MenuItem.Icon>
```

```
            <Image Source="/Icons/add.png"></Image>
        </MenuItem.Icon>
    </MenuItem>
    <MenuItem Header="_Apfel">
        <MenuItem.Icon>
            <Image Source="/Icons/apple.png"></Image>
        </MenuItem.Icon>
    </MenuItem>
    <MenuItem Header="_Ballon">
        <MenuItem.Icon>
            <Image Source="/Icons/balloon.png"></Image>
        </MenuItem.Icon>
    </MenuItem>
</MenuItem>
```

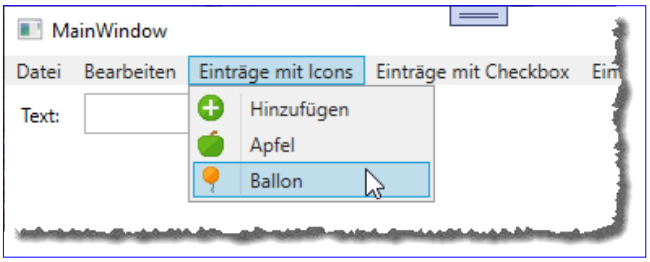


Bild 4: Menüs mit Icons

Die Icons werden dann wie in Bild 4 angezeigt.

Weitere Steuerelementtypen

Wie beim Ribbon gibt es noch weitere verwendbare Steuerelementtypen, aber bei weitem nicht so viele. Das meistverwendete neben dem herkömmlichen Menüeintrag dürfte der Trennstrich sein, der als Steuerelement die Bezeichnung **Separator** trägt. Den Trennstrich fügen Sie einfach zwischen zwei Menüeinträgen ein:

```
<MenuItem Header="_Schließen"></MenuItem>
<Separator />
<MenuItem Header="_Beenden"></MenuItem>
```

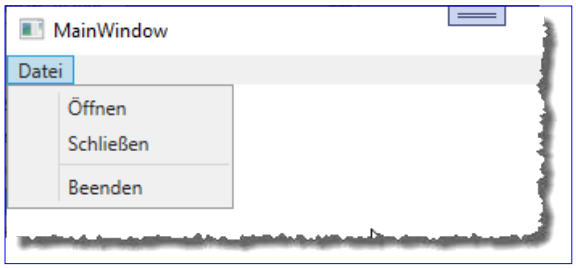


Bild 5: Menü mit Trennstrich

Er erscheint dann als waagerechter Strich zwischen den beiden anderen Einträgen (siehe Bild 5).

Menüeinträge mit Checkbox

Über Menüeinträge können Sie im kleinen Maßstab auch Optionen festlegen. Dazu bieten die **MenuItem**-Elemente die Eigenschaft **IsCheckable**. Hat diese Eigenschaft den Wert **True**, können Sie per Mausklick wie in Bild 6 einen Haken vor dem jeweiligen Eintrag einfügen. Um einen Eintrag direkt beim Anzeigen als angehakt darzustellen, weisen Sie außerdem dem Attribut **IsChecked** den Wert **True** zu. Hier ein Beispiel:

```
<MenuItem Header="Einträge mit Checkbox">
    <MenuItem Header="Erster Eintrag" IsCheckable="True" IsChecked="True"></MenuItem>
    <MenuItem Header="Zweiter Eintrag" IsCheckable="True"></MenuItem>
    <MenuItem Header="Dritter Eintrag" IsCheckable="True"></MenuItem>
</MenuItem>
```

Symbolleisten mit dem ToolBar-Steuerelement

Bei Anwendungen, die eine Menüleiste statt des Ribbons verwenden, finden Sie meist auch noch Symbolleisten vor. Davon gibt es im Gegensatz zur Menüleiste manchmal nicht nur eine, sondern mehrere. Symbolleisten zeichnen sich außerdem dadurch aus, dass Sie diese verschieben können und dass sie keine Untermenüs enthalten. Außerdem finden Sie die Befehle meist in Form einfacher Schaltflächen mit entsprechenden Symbolen vor, damit diese wenig Platz wegnehmen. Auch für eine WPF-Anwendung können Sie Symbolleisten hinzufügen. Wie das geht, zeigt der vorliegende Artikel.

Im Artikel **Das Menu-Steuerelement** (**www.datenbankentwickler.net/244**) haben wir bereits gezeigt, wie Sie eine Menüleiste zum Fenster einer Anwendung hinzufügen. Es gibt immer nur eine Menüleiste je Fenster, dem Sie ein oder mehrere Elemente in der obersten Ebene hinzufügen können. Diese stellen normalerweise per Mausklick eine aufklappbare Liste von Befehlen oder weiteren Untermenüs zur Verfügung. Bei den Symbolleisten ist es etwas anders: Sie können auch mehr als eine Symbolleiste hinzufügen, die nach dem Wunsch des Benutzers angeordnet werden kann. In den Beispielen zu diesem Artikel verwenden wir wieder ein übergeordnetes **DockPanel**-Element, dem wir eine Menüleiste und eine Symbolleiste hinzufügen.

Aufbau einer Symbolleiste

Bei der Menüleiste ist der Aufbau eindeutig: Es gibt ein **Menu**-Element, das im **DockPanel**-Element oben angedockt wird und das die benötigten Menüelemente enthält. Bei der Symbolleiste verwenden wir noch ein zusätzliches Element, nämlich das **ToolBarTray** in der obersten Ebene. Dieses ordnen Sie wie das **Menu**-Element dem **DockPanel**-Element unter (siehe Beispielprojekt, **MainWindow.xaml**, **Beispiel 1**):

```
<Window x:Class="MainWindow" ... Title="MainWindow" Height="350" Width="600">
    <DockPanel>
        <Menu DockPanel.Dock="Top">...</Menu>
        <ToolBarTray DockPanel.Dock="Top">
```

Das **ToolBarTray**-Element kann dann ein oder mehrerere **ToolBar**-Elemente enthalten, die wiederum **Button**-Elemente mit dem Befehlen bereithalten:

```
            <ToolBar>
                <Button Command="Cut" Content="Ausschneiden"></Button>
                <Button Command="Copy" Content="Kopieren"></Button>
                <Button Command="Paste" Content="Einfügen"></Button>
```

Zum Abschluss werden die geöffneten Elemente **ToolBar** und **ToolBarTray** wieder geschlossen und es folgt beispielsweise ein Grid-Element, das die übrigen Steuerelemente des Fensters enthält:

```
            </ToolBar>
```

```
        </ToolBarTray>
        <Grid>...</Grid>
    </DockPanel>
</Window>
```

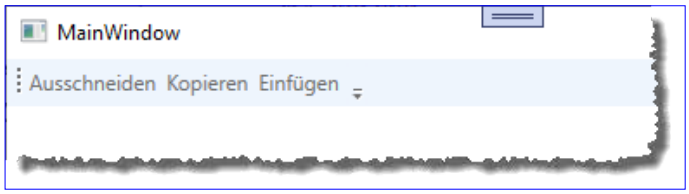


Bild 1: Fenster mit Symbolleiste

In Bild 1 sehen Sie die resultierende Symbolleiste. In diesem Fall haben wir einige eingebaute Befehle über das **Command**-Attribut verfügbar gemacht – **Cut**, **Copy** und **Paste**. Leider erkennt die Anwendung nicht, dass diese hier in einer Symbolleiste abgebildet werden sollen und zeigt keine Icons, sondern nur den Text an. Das können wir ändern!

Symbolleiste mit Icons

Wenn wir statt der Beschriftungen für die eingebauten Befehle **Cut**, **Copy** und **Paste** nur Icons anzeigen wollen, müssen wir diese zunächst zum Projekt hinzufügen. Dazu legen wir im Projektmappen-Explorer ein neues Verzeichnis namens **Icons** an und fügen diesem die benötigten Icons hinzu (siehe Bild 2).

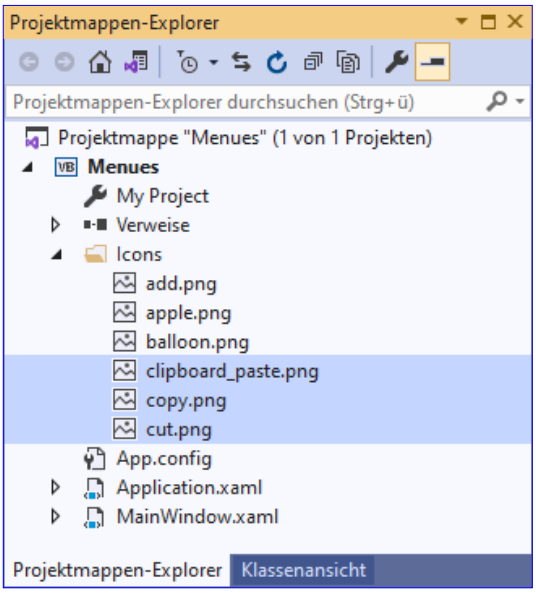


Bild 2: Hinzufügen von Icons zum Projekt

Danach bearbeiten wir die Steuerelemente der Symbolleiste in der XAML-Definition so, dass sie statt Text die Icons anzeigen. Dazu entfernen wir die **Content**-Attribute und fügen jeweils ein **Image**-Element hinzu. Dieses erhält für das Attribut **Source** den Pfad zu der Bilddatei im Projektverzeichnis (siehe Beispielprojekt, **MainWindow.xaml**, **Beispiel 2**):

```
<ToolBarTray DockPanel.Dock="Top">
    <ToolBar>
        <Button Command="Cut">
            <Image Source="Icons/cut.png"></Image>
        </Button>
        <Button Command="Copy">
            <Image Source="Icons/copy.png"></Image>
        </Button>
        <Button Command="Paste">
            <Image Source="Icons/clipboard_paste.png"></Image>
        </Button>
    </ToolBar>
</ToolBarTray>
```

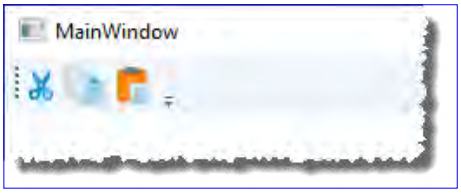


Bild 3: Symbolleiste mit Icons

Das Ergebnis sehen Sie in Bild 3. Interessanterweise werden die Schaltflächen mit den Icons im Gegensatz zu den reinen Text-Schaltflächen nicht in Abhän-

gigkeit vom Kontext als aktiviert oder deaktiviert gekennzeichnet. So sollte zum Beispiel die **Ausschneiden**-Schaltfläche nur als aktiv dargestellt werden, wenn im Textfeld darunter ein Text markiert ist. Das wollen wir später nachholen – erst einmal schauen wir uns an, wie wir eine weitere Symbolleiste hinzufügen und wie wir Icons und Texte auf einer Schaltfläche kombinieren können.

Eine zweite Symbolleiste hinzufügen

Zum Hinzufügen einer zweiten Symbolleiste legen Sie einfach ein weiteres **ToolBar**-Element im **ToolBarTray**-Element an. Je nachdem, ob die zweite Symbolleiste vor oder hinter der ersten erscheinen soll, fügen Sie das neue **ToolBar**-Element vor oder hinter einem bereits bestehenden **ToolBar**-Element ein (siehe Beispielprojekt, **MainWindow.xaml**, **Beispiel 3**):

```
<ToolBarTray DockPanel.Dock="Top">
    <ToolBar>
        ... Steuerelemente
    </ToolBar>
    <ToolBar>
        ... Steuerelemente
    </ToolBar>
</ToolBarTray>
```

In diesem Fall werden die beiden Symbolleisten direkt nebeneinander angezeigt (siehe Bild 4). Es gibt jedoch die Möglichkeit, diese auch untereinander auszugeben.

Bild 4: Zwei Symbolleisten nebeneinander

Der einfachste Weg ist, dass der Benutzer die Symbolleisten einfach per Drag and Drop so positioniert, wie er sich diese wünscht (siehe Bild 5). Beim nächsten Start der Anwendung werden die Symbolleisten allerdings wieder so wie im XAML-Code definiert ausgegeben. Wie also können wir die Symbolleisten direkt beim Starten untereinander anzeigen?

Bild 5: Zwei Symbolleisten untereinander

Symbolleistenposition relativ zu anderen Symbolleisten

Wenn Sie mehr als eine Symbolleiste verwenden, also mehr als ein **ToolBar**-Element innerhalb des **ToolBarTray**-Elements einfügen, werden diese wie oben beschrieben einfach in der gleichen Reihenfolge nebeneinander angeordnet. Wenn Sie die Reihenfolge ändern möchten, ohne die Reihenfolge im Code zu ändern, können Sie das mit der Eigenschaft **BandIndex** erledigen. **BandIndex** ist ein **0**-basierter Index (siehe Beispielprojekt, **MainWindow.xaml**, **Beispiel 4**). Wenn Sie also für das erste **ToolBar**-Element innerhalb des **ToolBarTray**-Elements den Wert **0** angeben, dann verbleibt es an der Position, die es vorher innehatte. Im folgenden Beispiel erhalten wir die Anordnung aus Bild 6:

```
<ToolBarTray DockPanel.Dock="Top">
    <ToolBar>
        <Button>Button in Toolbar 1</Button>
    </ToolBar>
```

```
    <ToolBar>
        <Button>Button in Toolbar 2</Button>
    </ToolBar>
    <ToolBar>
        <Button>Button in Toolbar 3</Button>
    </ToolBar>
</ToolBarTray>
```

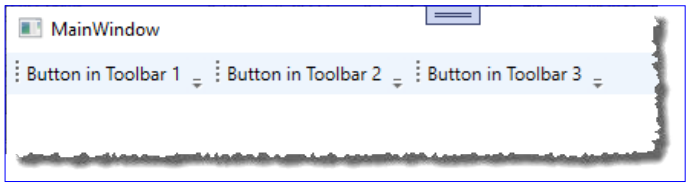


Bild 6: Standardreihenfolge

Stellen wir nun für das zuerst aufgeführte Element den Wert des Attributs **BandIndex** auf **2** ein, erhalten wir die Anordnung aus Bild 7.

```
<ToolBar BandIndex="2">
    <Button>Button in Toolbar 1</Button>
</ToolBar>
```

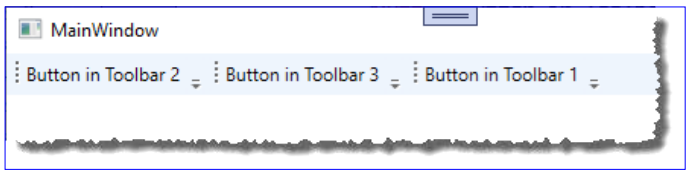


Bild 7: Per **BandIndex** angepasste Reihenfolge

Damit können Sie nun bereits die Reihenfolge innerhalb der gleichen Zeile anpassen. Nun wollen wir noch eines der **ToolBar**-Elemente in die zweite Zeile verschieben. Das erledigen wir mit dem Attribut **Band**. Behalten wir die übrigen Einstellungen bei und legen für das Attribut **Band** des ersten **ToolBar**-Elements den Wert **1** fest, landet diese Symbolleiste in der zweiten Zeile (siehe Bild 8):

```
<ToolBar Band="1">
    <Button>Button in Toolbar 1</Button>
</ToolBar>
```

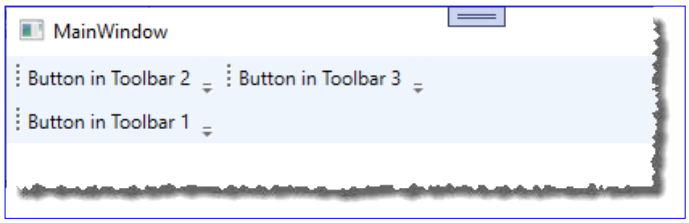


Bild 8: Verschieben einer Symbolleiste in eine andere Zeile mit dem **Band**-Attribut

Übrigens sind die Werte für **Band** und **BandIndex** relativ zu betrachten. Der Standardwert lautet **0**. Sie müssen aber nicht die Werte **0**, **1** und **2** verwenden, wenn die Symbolleisten in verschiedenen Zeilen landen sollen, sondern es funktioniert auch mit **2**, **4** und **6**.

Schaltflächen mit Icon und Text

Wollen Sie einer Symbolleiste nicht nur Schaltflächen mit Text oder Icon, sondern mit beidem hinzufügen, können Sie das mit den üblichen Techniken von WPF erledigen. Dabei fügen wir einfach ein **StackPanel**-Element als Inhalt des **Button**-Elements ein, dem wir dann mehrere weitere Steuerelemente hinzufügen können. Für unsere drei Schaltflächen sieht dies dann wie folgt aus (siehe Bild 9):

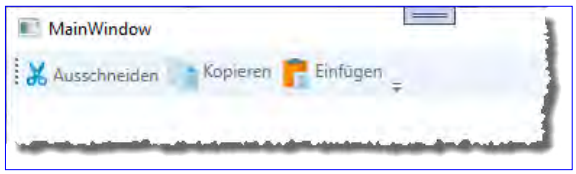


Bild 9: Symbolleiste mit Icons und Beschriftung

```
<ToolBar >
    <Button Command="Cut">
        <StackPanel Orientation="Horizontal">
```

Symbolleiste speichern und wiederherstellen

Mit dem ToolBar-Element können Sie einem Fenster auf einfache Weise eine Symbolleiste zuweisen. Je nach Umfang der Anwendung können so einige Symbolleisten zusammenkommen. Diese kann der Benutzer normalerweise selbst organisieren. Allerdings werden die Symbolleisten bei jedem Start wieder so hergestellt, wie es in der XAML-Definition vorgesehen ist. Ändert der Benutzer also die Position von Symbolleisten oder blendet er diese ein oder aus, halten diese Anpassungen immer nur bis zum nächsten Start der Anwendung. Außer natürlich, wir sehen eine Funktion vor, die den aktuellen Stand speichert und beim nächsten Start wiederherstellt.

Zu Beispielzwecken haben wir zwei **ToolBar**-Elemente zu einem **ToolBarTray** hinzugefügt. Diese haben wir wie folgt programmiert (Auszüge aus dem Code, den Sie im Modul **MainWindow.xaml** der Beispielanwendung finden):

```
<DockPanel>
    <ToolBarTray x:Name="tbt" DockPanel.Dock="Top">
        <ToolBar>
            <Button Command="Cut">
                <StackPanel Orientation="Horizontal">
                    <Image Source="Icons/cut.png"></Image>
                    <TextBlock Margin="3,0,0,0">Ausschneiden</TextBlock>
                </StackPanel>
            </Button>
            ...
        </ToolBar>
        <ToolBar>
            <Button>
                <Image Source="Icons/add.png"></Image>
            </Button>
            ...
        </ToolBar>
    </ToolBarTray>
</DockPanel>
```

Die Symbolleiste sieht nun wie in Bild 1 aus.

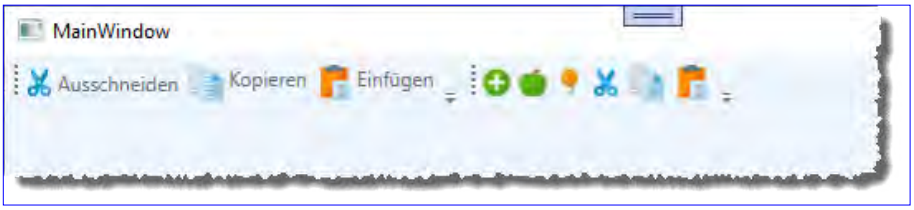


Bild 1: Beispielsymbolleisten

Der Benutzer hat nun die Möglichkeit, die Anordnung der Symbolleiste zu ändern. Zum Beispiel kann er die

Reihenfolge der beiden Symbolleisten vertauschen (siehe Bild 2).

Oder er verschiebt eine der beiden Symbolleisten in die zweite Zeile (siehe Bild 3).

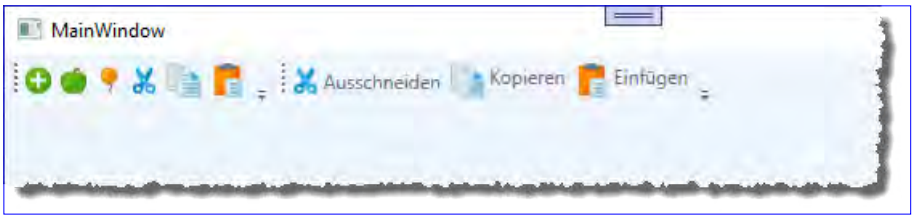


Bild 2: Vertauschte Beispielsymbolleisten

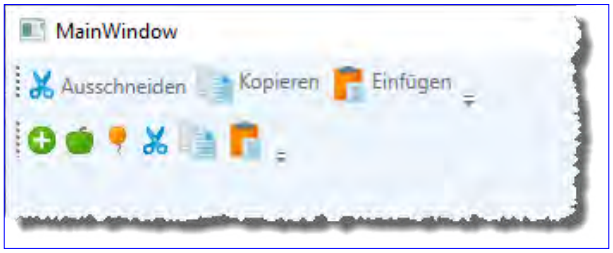


Bild 3: Beispielsymbolleisten in zwei Zeilen

Erste Idee: XAML-Definition des ToolBarTray-Elements speichern

Die erste Idee war, dass Änderungen der Position sich auch im Zustand der XAML-Definition der ToolBar-Elemente wiederspiegeln. Für diesen Fall wollten wir einfach die XAML-Definition des **ToolBarTray**-Elements speichern und diese beim nächsten Öffnen wiederherstellen. Die folgende Methode sollte durch ein Ereignis ausgelöst werden, das wir für das Element **Window** definieren:

```
<Window x:Class="MainWindow" ...  Title="MainWindow" Height="450" Width="800" Closing="Window_Closing">
```

Die dadurch ausgelöste Methode sieht wie folgt aus: Sie definiert ein **FileStream**-Objekt, das eine neue Datei namens **ToolBarTray.txt** erstellt und dann das ToolBarTray namens **tbt** mit der Variablen **objToolBarTray** referenziert. Danach nutzen wir die **Save**-Methode der **XamlWriter**-Klasse. Diese stellen wir über den folgenden Namespace bereit:

```
Imports System.Windows.Markup
```

Die **Save**-Methode erwartet einen Verweis auf das Root-Element des zu speichernden XAML-Inhalts sowie ein Ziel als Parameter. Nach dem Speichern schließen wir das **FileStream**-Objekt wieder:

```
Private Sub Window_Closing(sender As Object, e As ComponentModel.CancelEventArgs)
    Dim objToolBarTray As ToolBarTray
    Dim objFileStream As Stream
    objFileStream = New FileStream("ToolBarTray.txt", FileMode.Create)
    objToolBarTray = tbt
    XamlWriter.Save(objToolBarTray, objFileStream)
    objFileStream.Close()
End Sub
```

Nach dem Starten und Beenden der Anwendung finden wir im Anwendungsverzeichnis (hier **bin/debug**) eine Textdatei, die auch die Definition des gewünschten XAML-Bereichs enthält. Da wir zwischen Starten und Beenden noch keine Änderungen an der Konfiguration der **ToolBar**-Elemente vorgenommen haben, sieht der Code genau wie erwartet aus:

```
<ToolBarTray Name="tbt" DockPanel.Dock="Top" ...>
```

XML schreiben mit dem Document Object Model

Im Artikel »XML-Dokumente erstellen mit XmlWriter« haben wir beschrieben, wie Sie XML-Dokumente mit der XmlWriter-Klasse schreiben. Dort können Elemente nur sequenziell hinzugefügt werden. Die Methoden des Object Document Models erlauben eine wesentlich flexiblere Vorgehensweise. Dieser Artikel zeigt, wie Sie mit dem Document Object Model neue XML-Dokumente erstellen und Elemente der verschiedenen Typen hinzufügen und mit Daten füllen.

Beispieldaten

Die Beispiele aus dem ersten Teil des Artikels finden Sie im Beispielprojekt **XmlBeispiele** aus dem Download.

Erstellen eines neuen XML-Dokuments

Im Artikel **XML lesen mit dem Document Object Model** (**www.datenbankentwickler.net/237**) haben Sie neue XML-Dokument-Objekte erstellt und diese mit **Load** oder **LoadXml** befüllt. Nun wollen wir ein neues, leeres XML-Dokument erstellen und dieses speichern. Die Mindestanforderung ist dabei, dass das Dokument ein Stammelement enthält. Um das mit VB zu realisieren, benötigen wir wie in den vorherigen Artikeln einen Verweis auf den Namespace **System.Xml**:

```
Imports System.Xml
```

Danach können wir mit der folgenden Methode ein neues **XmlDocument** erstellen. Bevor wir dieses speichern können, fügen wir das Stammelement in zwei Schritten hinzu:

- Erstellen eines neuen Elements des Typs **XmlElement** mit der **CreateElement**-Methode des **XmlDocument**-Objekts und
- Hinzufügen dieses Elements mit der **AppendChild**-Methode an das XML-Dokument aus **objXml**.

Danach können wir das Dokument mit **Save** und dem gewünschten Dateipfad speichern:

```
Private Sub btnDokumentErstellenDOM_Click(sender As Object, e As RoutedEventArgs)
    Dim objXml As XmlDocument
    Dim objElement As XmlElement
    Dim strDocument As String
    strDocument = "C:\...\BestellverwaltungDOM.xml"
    objXml = New XmlDocument
    objElement = objXml.CreateElement("Bestellverwaltung")
    objXml.AppendChild(objElement)
    objXml.Save(strDocument)
End Sub
```

Das Dokument enthält nun schlicht folgenden Text:

```
<Bestellverwaltung />
```

Prolog hinzufügen

Für manche Anwendungen ist es wichtig, dass die XML-Datei einen Prolog enthält, also folgende Zeile:

```
<?xml version="1.0" encoding="utf-8"?>
```

Diese nennt man auch Processing Instruction. Und genau so ein Objekt gibt es auch im Namespace **System.Xml**. Um dieses zu erstellen, deklarieren wir eine Variable des Typs **XmlProcessingInstruction** sowie eine für den Inhalt dieses Objekts:

```
Dim objPI As XmlProcessingInstruction
Dim strPI As String
```

Dann schreiben wir den Text in **strPI** und erstellen mit der Methode **CreateProcessingInstruction** das **XmlProcessingInstruction**-Element. Der erste Parameter ist der Name, der hinter **<?** erscheinen soll, also **xml**, der zweite die Angabe der gewünschten Name-Wert-Paare. Danach hängen wir das Element wie herkömmliche Elemente mit der **AppendChild**-Methode an das Dokument an und speichern es anschließend:

```
strPI = "version=""1.0"" encoding=""utf-8"""
objPI = objXml.CreateProcessingInstruction("xml", strPI)
objXml.AppendChild(objPI)
...
objXml.Save(strDocument)
```

Das Ergebnis lautet, kombiniert mit den Anweisungen des ersten Beispiele, wie folgt:

```
<?xml version="1.0" encoding="utf-8"?>
<Bestellverwaltung />
```

Da wir eine solche Processing Instruction allen XML-Dokumenten voranstellen, gliedern wir die notwendigen Anweisungen in eine eigene Methode aus, die wie folgt aussieht:

```
Public Sub AddProcessingInstruction(ByVal objXml As XmlDocument)
    Dim objPI As XmlProcessingInstruction
    Dim strPI As String
    strPI = "version=""1.0"" encoding=""utf-8"""
    objPI = objXml.CreateProcessingInstruction("xml", strPI)
    objXml.AppendChild(objPI)
End Sub
```

Den Aufruf können wir dann so in die eigentliche Methode zum Erstellen des XML-Dokuments einstreuen:

```
AddProcessingInstruction(objXml)
```

Unterelemente hinzufügen

Unser XML-Dokument besteht nun gerade mal aus der Processing Instruction und dem Stammelement **Bestellverwaltung**. Weitere Elemente müssen wir zwingend dem Stammelement unterordnen. Dieses haben wir aber bereits mit **objElement** referenziert, sodass wir eigentlich direkt loslegen können. Allerdings machen wir uns direkt an dieser Stelle Gedanken über die Benennung der **XmlElement**-Variablen, denn wir werden ja vielleicht noch weitere verschachtelte Elemente wie Kunden, Bestellungen und Bestellpositionen einfügen wollen und wir benötigen für alle zumindest eine Objektvariable je Typ. Also ändern wir den Namen der Objektvariablen für das Stammelement in **objBestellverwaltung**. Die folgenden Elemente wollen wir dann **objKunden**, **objKunde**, **objBestellungen**, **objBestellung** und so weiter nennen. Als Erstes wollen wir nur das Unterelement **Kunden** hinzufügen. Dazu legen wir die Variable **objKunden** mit dem Typ **XmlElement** an.

Dieser weisen wir dann das mit der **CreateElement** von **objXml** erstellte Objekt mit dem Namen **Kunden** zu. Allerdings hängen wir das **Kunden**-Element nicht an das XML-Dokument aus **objXml** an, sondern an das Element aus **objBestellverwaltung**:

```
Private Sub btnDokumentErstellenDOM_Unterelemente_Click(sender As Object, e As RoutedEventArgs)
    ...
    Dim objKunden As XmlElement
    ...
    objBestellverwaltung = objXml.CreateElement("Bestellverwaltung")
    objXml.AppendChild(objBestellverwaltung)
    objKunden = objXml.CreateElement("Kunden")
    objBestellverwaltung.AppendChild(objKunden)
    objXml.Save(strDocument)
End Sub
```

Das Ergebnis sieht dann genau wie gewünscht aus:

```
<?xml version="1.0" encoding="utf-8"?>
<Bestellverwaltung>
  <Kunden />
</Bestellverwaltung>
```

Elemente mit Werten hinzufügen

Nun nähern wir uns den Elementen, die selbst Werte enthalten, also zum Beispiel das folgende:

```
  <Kunden>
    <Kunde>
      <Firma>André Minhorst Verlag</Firma>
    </Kunde>
  </Kunden>
```

Texte wie hier innerhalb des Elements **Firma** sind Elemente des Typs **XmlText**. Neben den übrigen neu hinzugekommenen Elementen deklarieren wir dieses entsprechend. Danach legen wir unterhalb des Elements aus **objKunden** weitere neue Elemente namens **objKunde** und **objFirma** an, beide des Typs **XmlElement**. Schließlich erstellen wir ein neues Element des Typs **XmlText** und weisen diesem mit dem Parameter **text** den gewünschten Inhalt zu. Dieses Objekt hängen wir an **objFirma** an:

```
Dim objKunde As XmlElement
Dim objFirma As XmlElement
Dim objText As XmlText
...
objKunde = objXml.CreateElement("Kunde")
objKunden.AppendChild(objKunde)
objFirma = objXml.CreateElement("Firma")
objKunde.AppendChild(objFirma)
objText = objXml.CreateTextNode("André Minhorst Verlag")
objFirma.AppendChild(objText)
objXml.Save(strDocument)
```

Damit hätten wir die Werkzeuge zum Zusammenstellen der Grundstruktur vollständig und können uns anderen Elementen zuwenden.

Attribute hinzufügen

Die Elemente **Kunde**, **Bestellung** und **Bestellposition** sollen jeweils um ein Attribut namens **ID** ergänzt werden:

```
<Kunde ID="123">
```

Dazu deklarieren wir eine Variable des Typs **XmlAttribute**:

```
Dim objID As XmlAttribute
```

AUSGABE 01-02/2021

DATENBANK ENTWICKLER

MAGAZIN FÜR DIE DATENBANKENTWICKLUNG MIT VISUAL STUDIO FÜR DESKTOP, WEB UND CO.

TOP-THEMEN:

BERICHTE	Reporting Services von A-Z	SEITE 45
ENTITY FRAMEWORK	Fluent API und DataAnnotations	SEITE 23
SQL SERVER	Schnelle Beispieldatenbank	SEITE 9
WPF	Standard- und Abbrechen-Schaltflächen	SEITE 3

André Minhorst Verlag

IsDefault: Standardschaltfläche festlegen

Unter WPF haben Sie verschiedene Möglichkeiten, eine Schaltfläche festzulegen, die beim Betätigen der Eingabetaste ausgelöst wird. Die Einfachste ist die über das Attribut »IsDefault« einer Schaltfläche. Stellen Sie dieses auf den Wert »True« ein, wird die normalerweise durch die Schaltfläche ausgelöste Ereignismethode auch beim Betätigen der Eingabetaste ausgelöst. Unter welchen Voraussetzungen das geschieht, zeigt dieser Artikel.

In Formularen oder Fenstern gibt es oft eine Schaltfläche, deren Funktion beim Betätigen der Eingabetaste ausgelöst werden soll. In vielen Fällen hat diese Schaltfläche die Beschriftung **OK** und soll beispielsweise das Fenster schließen.

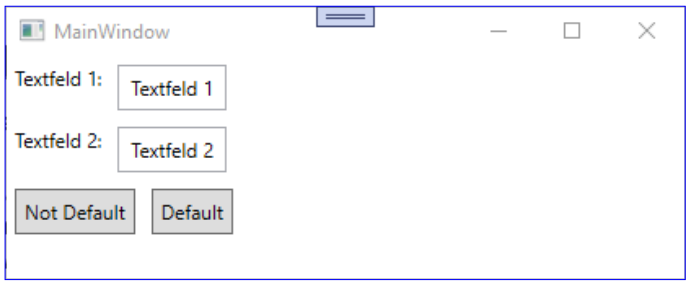


Bild 1: Default-Schaltfläche

Damit eine dafür vorgesehene Schaltfläche dies in einem WPF-Fenster erledigt, hat Microsoft die XAML-Eigenschaft **IsDefault** vorgesehen. Diese stellen Sie für die Schaltfläche, deren Ereignismethode für das Ereignis **Click** beim Betätigen der Eingabetaste ausgelöst werden soll, auf den Wert **True** ein:

```
<StackPanel Grid.Row="2" Orientation="Horizontal" VerticalAlignment="Top">
    <Button x:Name="btnNotDefault" Click="btnNotDefault_Click">Not Default</Button>
    <Button x:Name="btnDefault" IsDefault="True" Click="btnDefault_Click">Default</Button>
</StackPanel>
```

Das Beispiel sieht wie in Bild 1 aus. Wenn danach ein Steuerelement im gleichen Fenster den Fokus hat, das nicht auf das Ereignis **Enter** reagiert, wird die mit **IsDefault** markierte Schaltfläche ausgelöst. Dabei können beispielsweise die folgenden Fälle auftreten:

- Sie betätigen die Eingabetaste, wenn eine andere Schaltfläche oder ein ähnliches Steuerelement, für welches das **Click**-Ereignis implementiert ist, den Fokus hat. In diesem Fall wird die Ereignisprozedur für das Ereignis **Click** der Schaltfläche mit dem Fokus ausgelöst.

- Sie betätigen die Eingabetaste beispielsweise in einem **TextBox**-Steuerelement. Wenn dieses eine Ereignismethode für das Ereignis **Enter** enthält, wird dieses zuerst ausgeführt und danach das **Click**-Ereignis der Schaltfläche mit **IsDefault** = **True**.

- Wenn Sie jedoch für ein **TextBox**-Steuerelement die Eigenschaft **AcceptsReturn** auf **True** einstellen und die Eingabetaste betätigen, wenn dieses Steuerelement den Fokus hat, dann händelt dieses selbst das Betätigen der Eingabetaste, indem einen Zeilenumbruch bewirkt.

IsCancel: Abbrechen-Schaltfläche festlegen

Unter WPF gibt es verschiedene Wege, um zu definieren, was in einem Fenster beim Betätigen der Escape-Taste geschehen soll. Sie könnten beispielsweise die Ereignisse abfangen, die beim Drücken einer Taste ausgelöst werden und dabei prüfen, ob der Benutzer die Escape-Taste gedrückt hat. Meist jedoch gibt es eine Schaltfläche, welche die gleiche Funktion bereithalten soll wie die Escape-Taste – zum Beispiel das Schließen des Fensters über eine Abbrechen-Schaltfläche ohne Speichern der Daten. Wie Sie dies ganz einfach mithilfe des Attributs »IsCancel« erledigen, zeigt der vorliegende Artikel.

Theoretisch könnten Sie die Tastatureingaben des Benutzers abfangen, bis dieser die **Escape**-Schaltfläche betätigt und dann in einer entsprechenden Ereignismethode die gewünschten Anweisungen ausführen – zum Beispiel zum Schließen des aktuellen Fensters mit Verwerfen der getätigten Änderungen.

Bild 1: **Cancel**-Schaltfläche per **Escape** bedienen

Allerdings bietet WPF für das **Button**-Steuerelement ein Attribut, mit dem Sie dies noch einfacher erledigen können – ohne eine Zeile zusätzlichen Code. Während wir im folgenden Beispiel der Schaltfläche **btnOK** den Wert **True** für das Attribut **IsDefault** zuweisen, wodurch die dahinter stehende Ereignismethode beim Verwenden der Eingabetaste ausgelöst wird – mehr zu diesem Attribut siehe Artikel **IsDefault: Standardschaltfläche festlegen** (**www.datenbankentwickler.net/250**), verwenden wir für die Schaltfläche **cmdCancel** das Attribut **IsCancel**. Auch für dieses legen wir den Wert **True** fest. Das bewirkt, dass das Betätigen der **Escape**-Taste durch den Benutzer direkt an die Ereignismethode **Click** dieser Schaltfläche weitergeleitet wird:

```
<StackPanel Grid.Row="2" Orientation="Horizontal" VerticalAlignment="Top">
    <Button x:Name="btnOK" IsDefault="True" Click="btnOK_Click">OK</Button>
    <Button x:Name="btnCancel" IsCancel="True" Click="btnCancel_Click">Cancel</Button>
</StackPanel>
```

Dieser weisen Sie dann die entsprechende Anweisung zu – im folgenden Beispiel einfach nur die Anzeige eines Meldungsfensters:

```
Private Sub btnCancel_Click(sender As Object, e As RoutedEventArgs)
    MessageBox.Show("Cancel")
End Sub
```

Im Gegensatz zur Schaltfläche mit dem Wert **True** für das Attribut **IsDefault** wird die Taste mit **IsCancel** bei jeder Betätigung der **Escape**-Taste ausgelöst, auch wenn der Fokus sich gerade auf einer anderen Schaltfläche befindet.

SQL Server: Fehler bei Entwurfsänderungen

In letzter Zeit habe ich gelegentlich Änderungen an den Tabellen verschiedener SQL Server-Datenbanken vorgenommen. Einige Änderungen dienten der Erweiterung um neue Felder, andere haben Primärschlüssel, Indizes, Beziehungen und andere Elemente angepasst. Die betroffenen Tabellen enthalten teilweise sehr viele Daten, sodass SQL Server recht viel Zeit beanspruchte, die Tabellen zu ändern. Teilweise gab es Fehler, weil Zeitlimits überschritten wurden. Teilweise konnten aber auch neue Beziehungen oder Indizes nicht angelegt werden, weil die Integrität der Daten nicht gegeben war. Dieser Artikel zeigt, welche Einstellungen Sie dann ändern können und welche Ursachen es gibt, wenn all dies nichts hilft.

SQL Server Entwurfsänderungen mit Timeout

Manchmal sind Änderungen am Entwurf einer Tabelle mit sehr vielen Datensätzen für den SQL Server sehr aufwendig durchzuführen. Oft reicht die Zeit nicht aus, weil zum Beispiel eine neue, temporäre Tabelle erzeugt, die Datensätze kopiert und die alte Tabelle gelöscht sowie die neue in die alte umbenannt werden muss. In solchen Fällen erscheint die Meldung aus Bild 1.

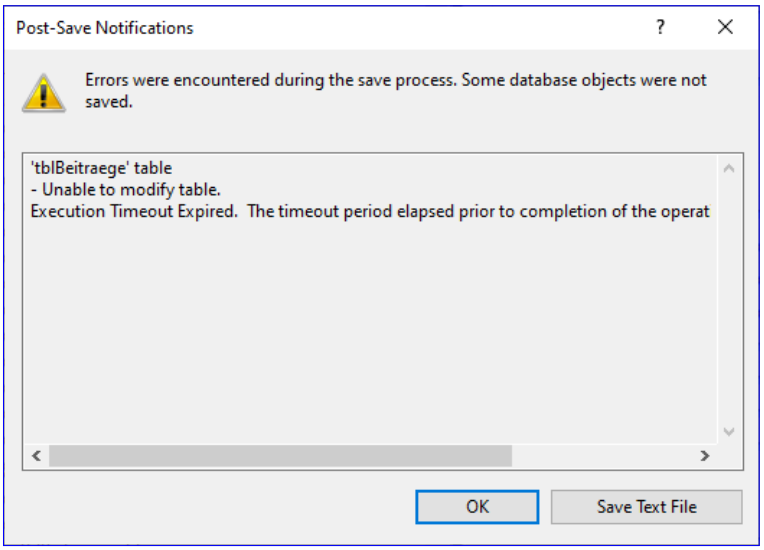


Bild 1: Timeout beim Ändern eines Feldnamens

Das Zeitlimit können Sie über die Optionen des SQL Server Management Studios einstellen. Dazu öffnen Sie den Options-Dialog mit dem Befehl **Tools|Options**. Hier finden Sie unter **Designers|Table and Database Designers** die Option **Override connection string time-out value for table designer updates**, der aktiviert sein sollte, und darunter die Einstellung **Transaction time-out after**, die standardmäßig 30 Sekun-

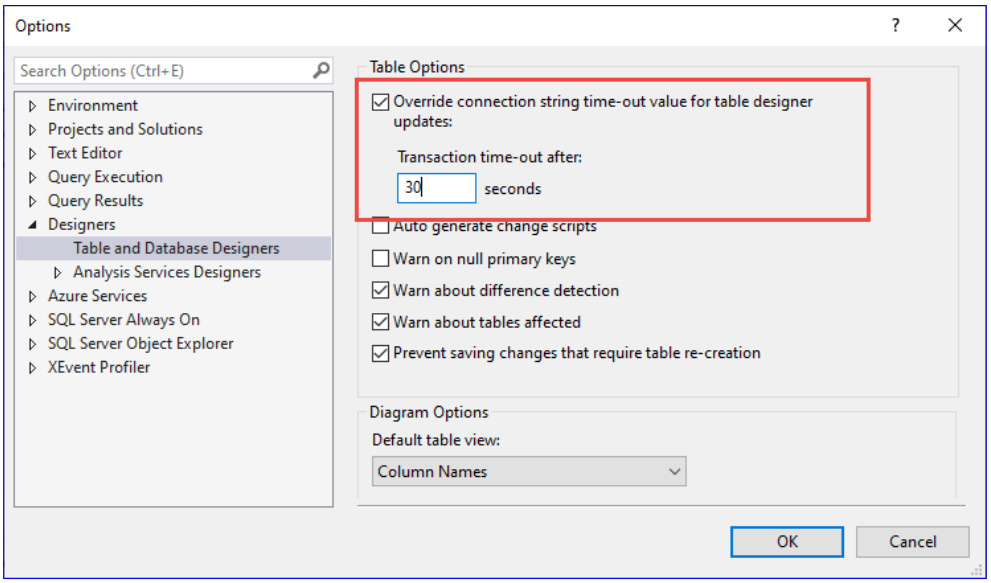


Bild 2: Einstellungen für den Timeout

den beträgt. Wenn Sie also Timeouts haben, setzen Sie einfach diese Einstellung auf einen höheren Wert (siehe Bild 2).

Entwurfsänderungen, die das Neuerstellen der Tabelle erfordern

Wenn Sie eine der folgenden Aktionen im Entwurf einer Tabelle durchführen und diese Änderungen dann speichern wollen, erhalten Sie eventuell einen weiteren Fehler, den Bild 3 zeigt:

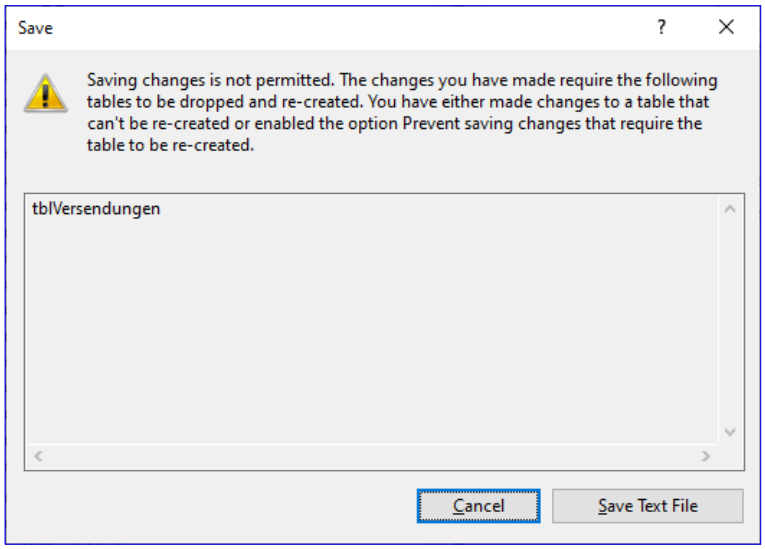


Bild 3: Fehler, weil die Tabelle nicht neu erstellt werden kann

- Sie ändern die Feldeigenschaft **Allow Nulls**.
- Sie ändern die Reihenfolge von Feldern in der Tabelle.
- Sie ändern den Datentyp eines Feldes.
- Sie fügen ein neues Feld hinzu.

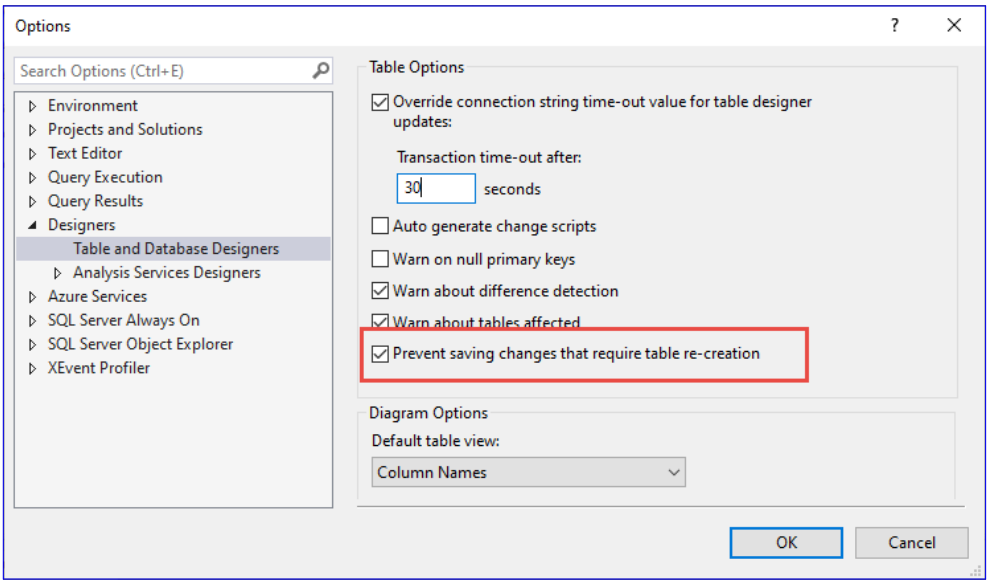


Bild 4: Einstellung zum Freigeben der Neuerstellung von Tabellen beim Ändern

Warum erhalten Sie den Fehler »eventuell«? Weil es davon abhängt, ob die Einstellungen des SQL Servers das Speichern von Änderungen, die ein Neuerstellen der Tabelle erfordern, erlauben. Diese Einstellung finden Sie, wenn Sie im SQL Server Managgement Studio mit Tools|Options den Options-Dialog öffnen. Hier wechseln Sie links zum Bereich **Designers|Table and Database Designers** und finden dort die Option **Prevent saving changes that require table re-creation** (siehe Bild 4). Wenn diese aktiviert ist, können die oben genannten Änderungen nicht gespeichert werden. Um das zu erledigen, müssen Sie die Option deaktivieren.

SQL Server Entwurfsänderungen für Fremdschlüsselfelder

In einer Datenbank habe ich festgestellt, dass eine Beziehung zwischen dem Fremdschlüsselfeld **KundeID** der Tabelle **tblVersendungen** und der Tabelle **tblKunden** noch gar nicht angelegt war. Also habe ich diese nachträg-

AdventureWorks: Schnelle Beispieldatenbank

Vielleicht benötigen Sie zu irgendeinem Zweck einmal eine SQL Server-Beispieldatenbank. Microsoft stellt eine solche zur Verfügung: Diese heißt AdventureWorks und steht in verschiedenen Versionen zum kostenlosen Download bereit. Dieser Artikel zeigt, wo Sie diese Beispieldatenbank finden und wie Sie diese aus einem Backup auf ihrem Rechner wiederherstellen.

Download der AdventureWorks-Beispieldatenbank

Die **AdventureWorks**-Beispieldatenbank finden Sie zum Download, wenn Sie bei Google **AdventureWorks** eingeben. Hier finden Sie drei verschiedene Versionen mit den Bezeichnungen OLTP (Online Transaction Processing), Data Warehouse und Lightweight. Wir verwenden hier die Lightweight-Version. Die Datenbank kommt außerdem für verschiedene SQL Server-Versionen. Wählen Sie einfach die für Ihre Version des SQL Servers passende Version der Beispieldatenbank aus.

Wir haben so für SQL Server 2019 die Datei **AdventureWorksLT2019.bak** heruntergeladen.

Wiederherstellung der AdventureWorks-Beispieldatenbank

Um die Datenbank aus der Sicherungskopie wiederherzustellen, speichern wir diese zunächst in dem Verzeichnis, in das SQL Server standardmäßig Backup-Dateien ablegt. Dieses Verzeichnis lautet unter Windows 10 beispielsweise wie folgt:

```
C:\Program Files\Microsoft SQL Server\MSSQL15.MSSQLSERVER\MSSQL\Backup
```

Nun haben Sie die Möglichkeit, die Datenbank über die Benutzeroberfläche des SQL Server Management Studios wiederzustellen. Dazu öffnen Sie SQL Server Management Studio und melden sich an der SQL Server-Instanz an, in der Sie die Datenbank wiederherstellen möchten.

Wiederherstellen per Benutzeroberfläche

Anschließend klicken Sie mit der rechten Maustaste auf den Eintrag **Databases** des Objekt-Explorers des SQL Server Management Studios. Hier finden Sie den Kontextmenü-Befehl **Restore Database...** und betätigen diesen (siehe Bild 1).

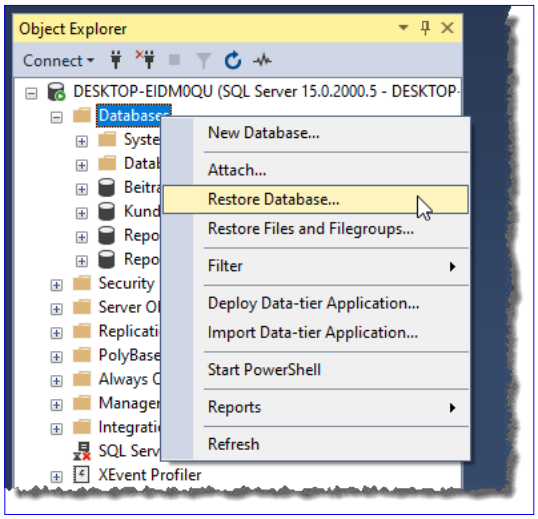


Bild 1: Kontextmenü-Eintrag zum Wiederherstellen einer Datenbank

Im nun erscheinenden Dialog **Restore Database** wählen Sie im rechten Bereich unter **Source** die Option **Device** aus und klicken auf die rechts daneben befindliche Schaltfläche mit den drei Punkten (**...**).

Danach erscheint der Dialog **Select backup device**, in dem Sie wiederum über die Schaltfläche **Add** den Dialog zum Auswählen der Backup-Datei auswählen (siehe Bild 2).

EDM mit vorhandener SQL Server-Datenbank

In einigen der vorherigen Ausgaben haben wir uns mit dem Erstellen eines Entity Data Models auf Basis von Access-Datenbanken konzentriert. Dabei haben wir aus dem Datenmodell der Access-Datenbank ein Entity Data Model erstellt, mit dem dann die SQL Server-Datenbank angelegt werden konnte. Nun wollen wir den vermutlich nicht so seltenen Fall betrachten, dass bereits eine SQL Server-Datenbank existiert. Dabei zeigen wir, wie Sie das Entity Data Model erstellen und damit auf die in der SQL Server-Datenbank enthaltenen Daten zugreifen können – und wie Sie die Namen von Entitäten und Eigenschaften angleichen, wenn die in den Tabellen und Feldern verwendeten Namen nicht passen.

EDM manuell oder per Assistent erstellen?

Bei unseren Techniken zum Erstellen eines Entity Data Models auf Basis des Datenmodells einer Access-Datenbank haben wir gesehen, dass der entstehende Code überschaubar und von der Anzahl der Tabellen und Felder abhängig ist. Je nachdem, ob die Anwendung alle Tabellen des Datenmodells der Datenbank verwenden soll oder vielleicht nur einige wenige, kann es eine Alternative sein, die Objekte manuell zu erstellen.

Ein wichtiger Faktor dabei ist, ob die Tabellen und Felder so benannt sind, dass Sie diese direkt in das Entity Data Model übernehmen können. Dabei ist es zum Beispiel eher untypisch, eine Entität mit **tblKunde** zu bezeichnen, auch wenn die Tabelle **tblKunden** heißt. Schöner wäre es, wenn wir hier **Kunde** statt **tblKunden** verwenden könnten. Wie wir gleich sehen werden, übernimmt Visual Studio bei Verwendung des Assistenten immer die Namen der Tabellen und Felder für die Entitäten und ihre Eigenschaften. Wenn wir also alle oder auch nur einige Tabellen und ihre Felder als Grundlage für die Erstellung eines Entity Data Models verwenden, müssen wir noch

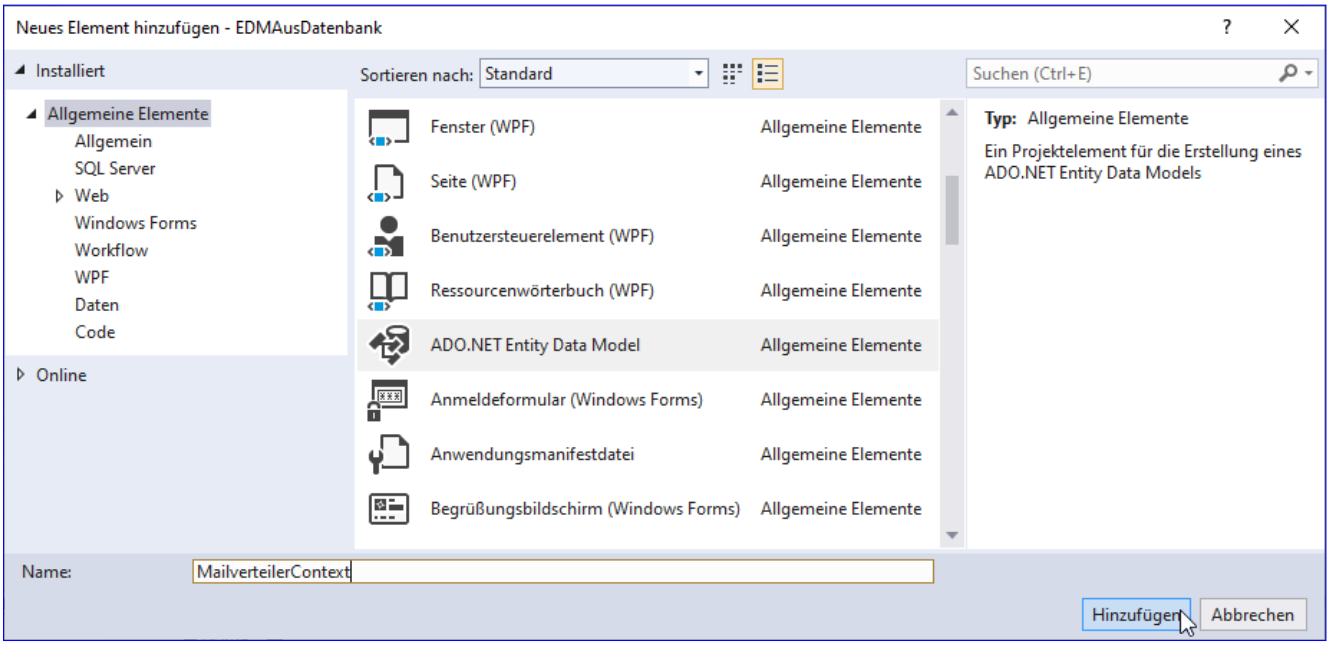


Bild 1: ADO.NET Entity Data Model hinzufügen

einen Weg finden, wie wir eine Art Mapping erstellen. Dieses soll dafür sorgen, dass wir die Tabellen- und Feldnamen wie **tblKunden** oder **KundeID** in der Datenbank beibehalten und gleichzeitig Entitätsnamen wie **Kunde** oder eine Bezeichnung wie **ID** für ein Schlüsselfeld verwenden können.

Entity Data Model per Assistent erstellen

Wenn wir das Entity Data Model per Assistent erstellen, legen wir dazu zunächst ein neues Projekt des Typs **WPF-App (.NET Framework)** für Visual Basic an. Danach fügen wir ein neues Element des Typs **ADO.NET Entity Data Model** zum Projekt hinzu. Dazu wählen Sie den Menübefehl **Projekt|Neues Element hinzufügen...** aus und wählen im nun erscheinenden Dialog den Eintrag **ADO.NET Entity Data Model** aus. Außerdem legen Sie unter **Name** die gewünschte Bezeichnung fest, in diesem Fall **MailverteilerContext** (siehe Bild 1). Der Hintergrund ist, dass wir in einem weiteren Artikel in der folgenden Ausgabe eine Anwendung mit einem Mailverteiler programmieren, der nur wenige Tabellen der SQL Server-Datenbank nutzen soll – und dessen Entity Data Model wir hier gleich vorbereiten.

Danach erscheint der Dialog **Assistent für Entity Data Model**, in dem wir den Eintrag **Code First aus Datenbank** auswählen (siehe Bild 2).

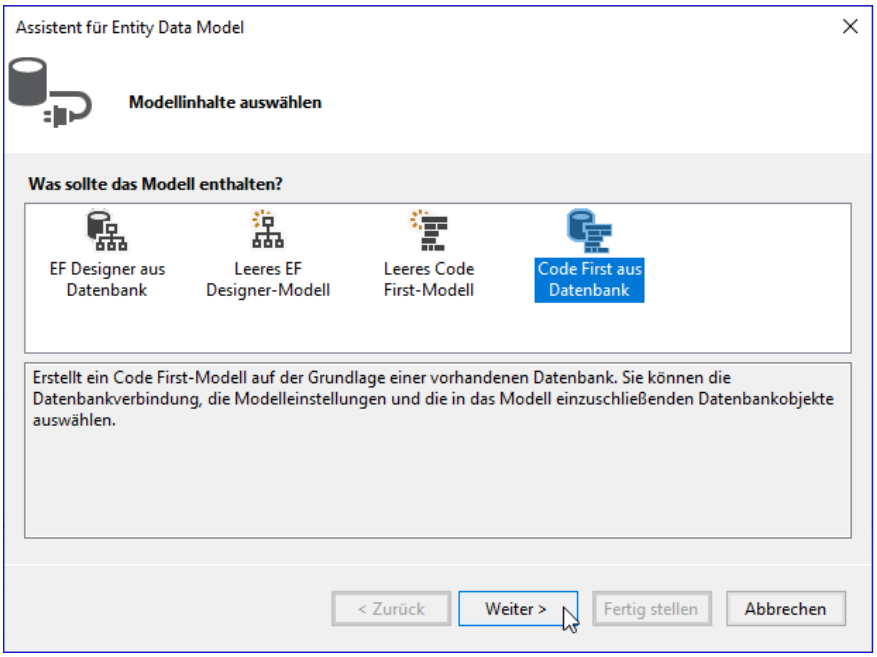


Bild 2: Auswahl des Eintrags **Code First aus Datenbank**

Verbindung einstellen

Im nächsten Schritt klicken Sie im Dialog **Wählen Sie Ihre Datenverbindung aus** auf

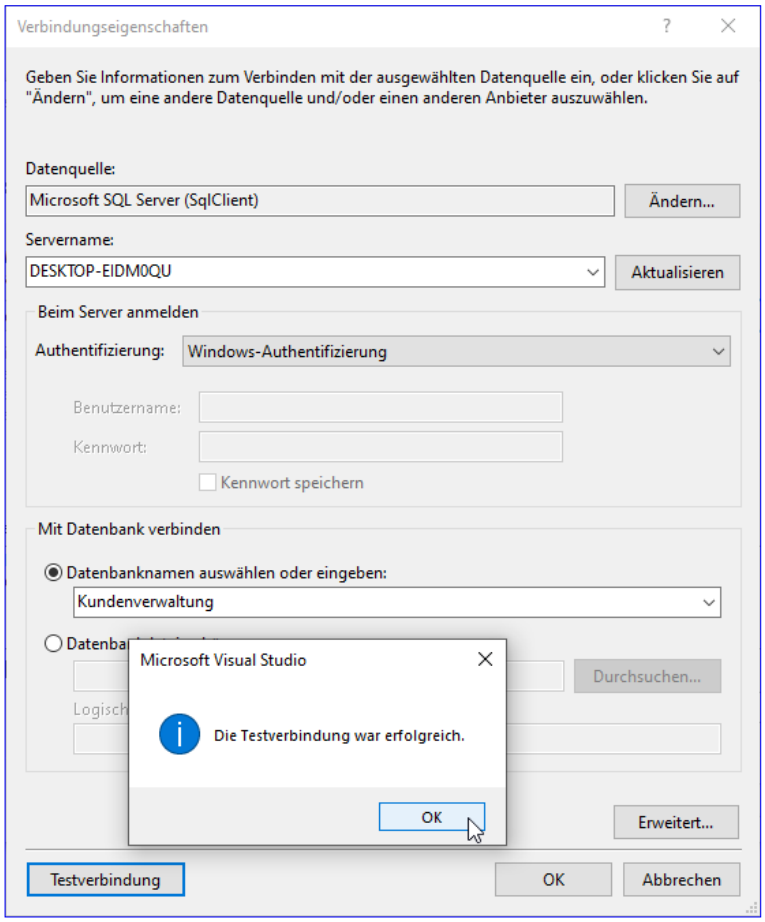


Bild 3: Einstellen der Verbindungseigenschaften

den Befehl **Neue Verbindung...** (außer, Sie haben dort zuvor schon einmal die gewünschte Verbindung angegeben – dann können Sie diese direkt auswählen). Damit erscheint der Dialog **Verbindungseigenschaften**, wo Sie zuerst den Namen des SQL Servers eingeben oder diesen aus dem SQL Server Management Studio herauskopieren und einfügen, da das Einlesen gelegentlich etwas länger dauert. Wenn Sie den Servernamen eingegeben haben, stellen Sie noch die Authentifizierungsart ein und hier gegebenenfalls noch die Zugangsdaten. Außerdem können Sie nun die Datenbank auswählen. Nach einem Test mit einem Klick auf die Schaltfläche **Testverbindung** und der Bestätigung, dass die Verbindung funktioniert, beenden Sie den Dialog mit **OK** (siehe Bild 3).

Verbindungszeichenfolge prüfen

Nach dem Schließen kehren wir zum Dialog-Schritt **Wählen Sie Ihre Datenverbindung aus** zurück. Dieser zeigt Ihnen nun die zusammengestellte Verbindungszeichenfolge an. Außerdem können Sie hier festlegen, unter welchem Namen diese Einstellungen in der Datei **App.Config** hinzugefügt werden soll (siehe Bild 4).

Datenbankobjekte auswählen

Der nächste Schritt bietet die einzelnen Tabellen und Views der

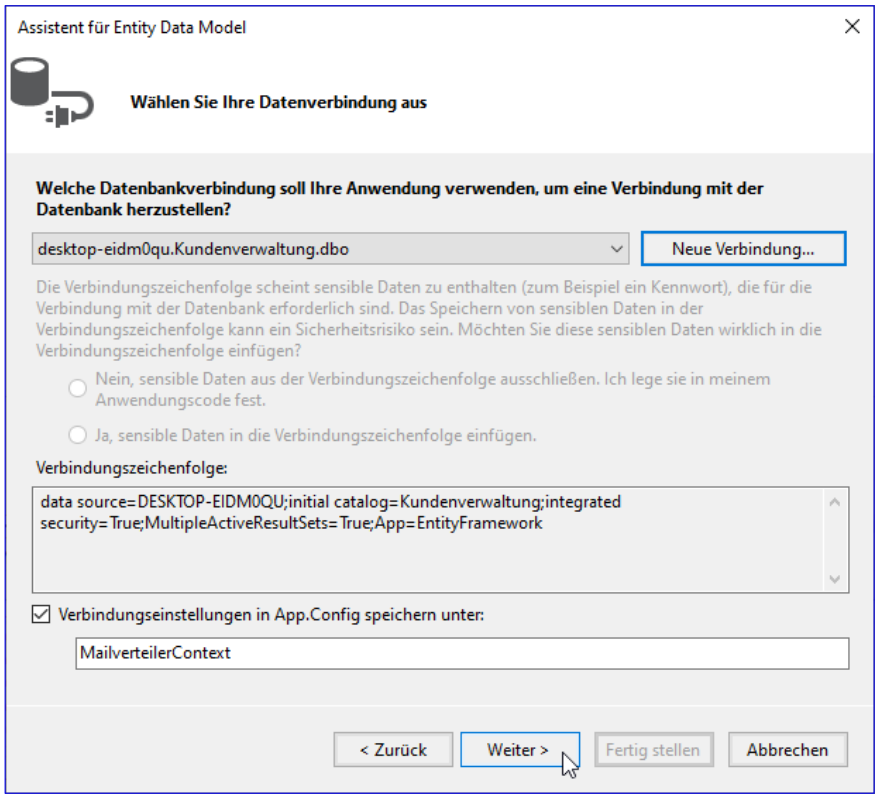


Bild 4: Einstellen weiterer Informationen

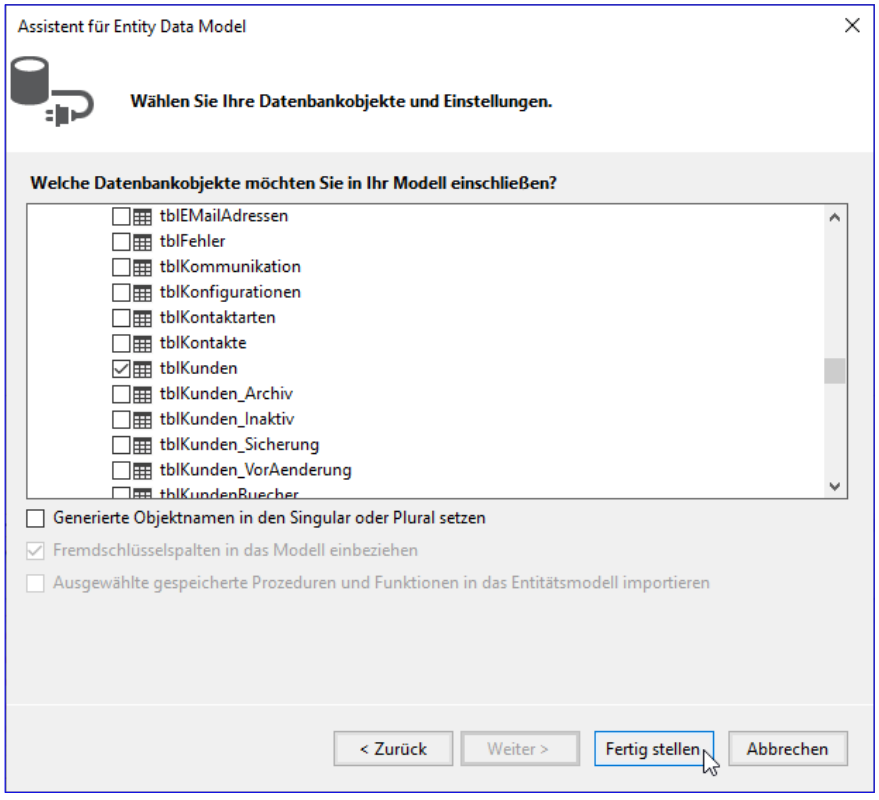


Bild 5: Auswahl der zu verwendenden Tabellen

SQL Server-Datenbank zur Auswahl an. Hier selektieren wir zuerst einmal die beiden Tabellen **tblAnreden** und **tblKunden** (siehe Bild 5). Die Option **Generierte Objektnamen in den Singular oder Plural setzen** sorgt dafür, dass für die Ermittlung der **DbSet**-Elemente, also der Auflistungselemente für die Entitäten, jeweils noch ein **s** an den Tabellennamen angehängt wird. Für die Tabelle **tblKunden** würde der Assistent dann ein **DbSet** namens **tblKundens** erstellen. Das ist noch weniger funktional, als die **DbSet**-Elemente und die Entitäten gleichermaßen mit **tblKunden** zu benennen.

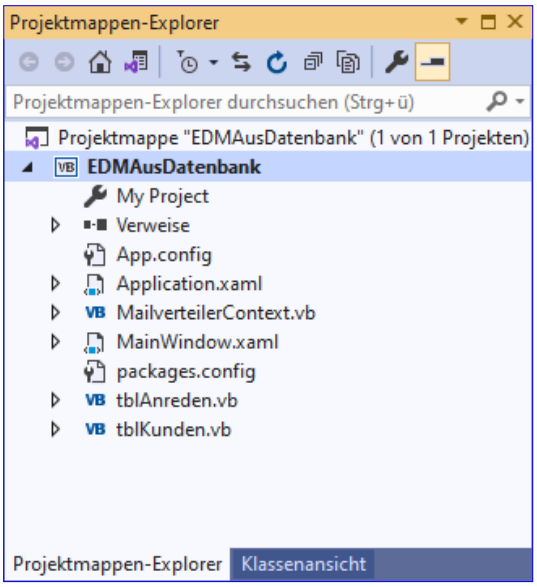


Bild 6: Die neuen Objekte im Projektmappen-Explorer

Erstellen des Entity Data Models

Nach einem Klick auf die Schaltlfäche **Fertigstellen** arbeitet Visual Studio ein wenig und kurz darauf erscheinen einige neue Einträge im Projektmappen-Explorer beziehungsweise wurden angepasst (siehe Bild 6). Die neuen Objekte heißen:

- **MailverteilerContext.vb**: Enthält die Definition der **DbSet**-Elemente sowie eine Methode namens **OnModelCreating**, die weitere Eigenschaften für verschiedene Felder der Tabelle **tblKunden** einstellt.
- **tblAnreden.vb**: Entitätsklasse für die Tabelle **tblAnreden**
- **tblKunden.vb**: Entitätsklasse für die Tabelle **tblKunden**

Schließlich finden wir in der Datei **App.config** noch ein paar neue Zeilen, welche das Entity Framework definieren sowie die Angabe der Verbindungszeichenfolge:

```
<?xml version="1.0" encoding="utf-8"?>
<configuration>
  ...
  <entityFramework>
    <defaultConnectionFactory type="System.Data.Entity.Infrastructure.LocalDbConnectionFactory, EntityFramework">
    ...
  </entityFramework>
  <connectionStrings>
    <add name="MailverteilerContext" connectionString="data source=DESKTOP-EIDM0QU;initial catalog=Kundenverwaltung;integrated security=True;MultipleActiveResultSets=True;App=EntityFramework" providerName="System.Data.SqlClient" />
  </connectionStrings>
</configuration>
```

Außerdem hat der Assistent einige Verweise auf Bibliotheken wie **EntityFramework** et cetera hinzugefügt.

Klassen mit den Namen von Entitätsklassen und Eigenschaften

Nun haben wir schon weiter oben erläutert, dass der Assistent aus Tabellennamen mit Präfixen, wie wir diese verwenden, keine geeigneten Namen für die zu erzeugenden Entitätsklassen ableiten kann. Bei den Feldnamen ist es weniger problematisch, aber auch hier möchten Sie vielleicht Änderungen vornehmen.

Schauen wir uns zunächst die Entitätsklasse für die Tabelle **tblAnreden** an, weil diese wegen der wenigen Eigenschaften recht übersichtlich ist:

```
<Table("tblAnreden")>
Partial Public Class tblAnreden
    Public Sub New()
        tblKunden = New HashSet(Of tblKunden)()
    End Sub

    <Key>
    Public Property AnredeID As Integer

    <StringLength(255)>
    Public Property Anrede As String

    <StringLength(255)>
    Public Property Briefanrede As String

    <StringLength(255)>
    Public Property Adressanrede As String

    Public Overridable Property tblKunden As ICollection(Of tblKunden)
End Class
```

Die Klasse **MailverteilerContext.vb** definiert für die beiden Entitätsklassen **tblAnreden** und **tblKunden** jeweils ein **DbSet**-Objekt gleichen Namens, was auch nicht gerade funktional ist:

```
Partial Public Class MailverteilerContext
    Inherits DbContext
    ...
    Public Overridable Property tblAnreden As DbSet(Of tblAnreden)
    Public Overridable Property tblKunden As DbSet(Of tblKunden)
    ...
End Class
```

Code First Mapping per Fluent API

Wenn Sie ein Code First-Entity Data Model entwerfen, gibt es bestimmte Konventionen, die standardmäßig greifen. So heißen Entitäten wie der Singular der zugrunde liegenden Tabellennamen oder der Primär- und Fremdschlüssel werden aus Feldern abgeleitet, welche die Zeichenkette ID enthalten – gegebenenfalls kombiniert mit dem Entitätsnamen. Wenn Sie mit einem Datenmodell daherkommen, dessen Tabellen das Präfix »tbl« mitbringen, wollen Sie dieses nicht in den Entitätsnamen wiederfinden und gegebenenfalls möchten (oder müssen) Sie auch noch Feldnamen ändern und diese anschließend mappen. Eine Möglichkeit dazu finden Sie in den Methoden der Fluent Api, die wir in diesem Artikel beschreiben.

Warum Mapping zwischen Datenmodell und Entitäten?

Wenn Sie ein Entity Data Model auf Basis des Datenmodells einer Datenbank erstellen oder andersherum, ist immer eine bestimmte Art von Mapping erforderlich. Im Optimalfall sind die Tabellen- und Feldnamen genau so eingestellt, dass diese den DbSet- und Entitätsnamen entsprechen.

Sobald Sie aber ein Datenmodell erhalten, das beispielsweise Tabellennamen mit dem Präfix **tbl** (wie **tblArtikel**) enthält und wo die Feldnamen vielleicht nicht den Wünschen des Entwicklers für die Eigenschaften der Entitäten entsprechen, müssen Sie diese auf irgendeine Weise anpassen. Das Datenmodell anzupassen wäre toll, aber in manchen Fällen greifen noch andere Anwendungen auf die Datenbank zu, sodass Sie die Namen von Tabellen und Feldern nicht einfach ändern können.

Also stellt das Entity Framework Möglichkeiten bereit, das Mapping zwischen den Tabellen und Feldern auf der einen und den DbSets und Eigenschaften auf der anderen Seite anzupassen.

Dabei kann das Mapping für verschiedene Zwecke genutzt werden:

- Wenn Sie ein Entity Data Model auf Basis einer bestehenden Datenbank etwa vom SQL Server für Code First erstellt haben und die Auflistungen, Entitätsklassen und ihre Eigenschaften so anpassen wollen, dass es Ihren Programmierkonventionen entspricht, auch wenn die Definition der Tabellen und Felder anders lautet oder

- wenn Sie ein Entity Data Model als Code First-Model programmiert haben und dann per Migration-Funktionalität daraus die Datenbank erzeugen wollen. Wie dies gelingt, schauen wir uns an einigen Beispielen in den folgenden Abschnitten im Detail an.

Mapping mit der Fluent API

Die Fluent API des Entity Frameworks bietet Methoden, die genau an einer Stelle innerhalb des Entity Data Models ausgeführt werden: in der Überschreibung der Methode **OnModelCreating** der **DbContext**-Klasse. Entity Framework ruft diese Methode beim Erstellen des Entity Data Models auf und führt die dort angegebenen Anweisungen zum Anpassen des Mappings aus.

Beispielprojekt

Das Beispielprojekt erstellen wir als neues Projekt des Typs **WPF-App (.NET Framework)** für Visual Basic. Damit wir die nun zu erstellenden Elemente in eine Datenbank übertragen können, fügen wir dem Projekt ein neues Element des Typs **ADO.NET Entity Data Model** hinzu und nennen dieses **FluidAPIContext** (siehe Bild 1).

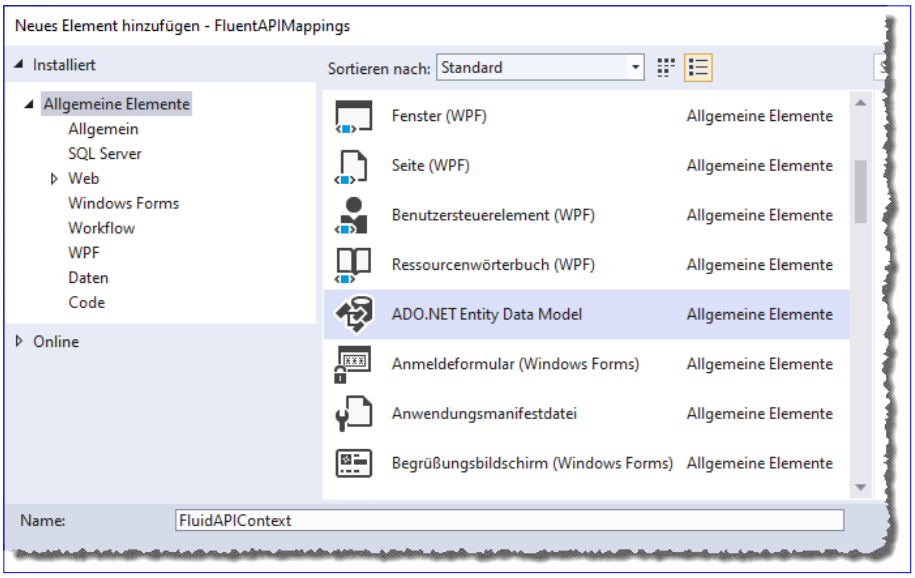


Bild 1: Entity Data Model hinzufügen

Dabei verwenden wir als Vorlage **Leeres Code First-Modell** (siehe Bild 2). Nach einem Klick auf die Schaltfläche **Fertigstellen** erhalten wir im Projektmappen-Explorer ein neues Element namens **FluidAPIContext.vb** sowie ein paar neue Einträge in der Datei **App.config** – zum Beispiel die Verbindungszeichenfolge zum Erstellen beziehungsweise für den Zugriff auf die Datenbank für diese Lösung.

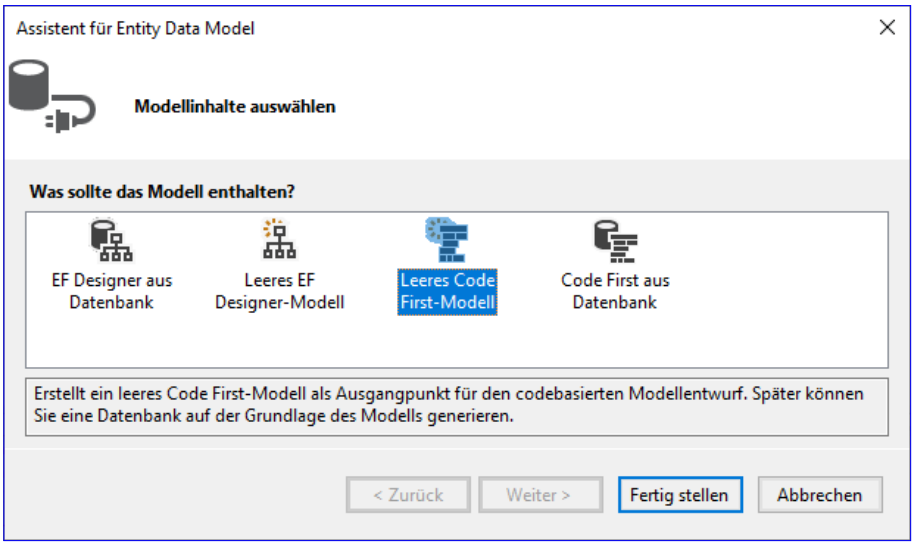


Bild 2: Neues Entity Data Model als leeres Code First-Modell

Einfache Tabelle erstellen

Zu einer einfachen Tabelle gehört ein eindeutiges Primärschlüsselfeld mit Autowert-Funktion sowie die benötigten Felder. Wir beginnen mit einer Entität, auf deren Basis eine Tabelle namens **Anreden** entstehen soll. Diese soll die Felder **ID** und **Name** enthalten. Dazu legen wir eine neue Klassendatei namens **Anrede.vb** an. Der Code der Klasse lautet vorerst:

```
Public Class Anrede
    Public Property ID As Integer
    Public Property Name As String
End Class
```

Für dieses Element fügen wir der **DbContext**-Klasse eine Auflistung hinzu:

```
Public Class FluidAPIContext
    Inherits DbContext
    Public Sub New()
        MyBase.New("name=FluidAPIContext")
    End Sub
    Public Overridable Property Anreden As DbSet(Of Anrede)
End Class
```

Um daraus die Datenbank mit der ersten Tabelle zu erstellen, benötigen wir die Paket-Manager-Konsole, die wir mit dem Befehl **Extras|NuGet-Paketmanager|Paket-Manager-Konsole** anzeigen. In der Konsole setzen Sie nacheinander die folgenden Anweisungen ab:

- **Enable-Migrations**: Aktiviert die Migrationen vom Entity Data Model zu einer SQL Server-Datenbank. Dies fügt den Ordner **Migrations** mit der Datei **Configuration.vb** zum Projekt im Projektmappen-Explorer hinzu.

- **Add-Migration Init**: Erstellt die Befehle für die initiale Migration, die beispielsweise in einer Datei namens **202103111053588_Init.vb** im Ordner **Migrations** angelegt wird. Die hier gespeicherten Befehle werden für das Erstellen beziehungsweise Anpassen der Datenbank verwendet.

- **Update-Database**: Führt die in der mit **Add-Migration** erstellten Klasse enthaltenen Befehle aus, um die Datenbank erstmalig zu erstellen und die angegebene Tabelle anzulegen.

Dies erstellt die neue Datenbank und eine Tabelle, die wir uns im SQL Server Management Studio anschauen können (den Zielserver und den Namen der Datenbank können Sie der Verbindungszeichenfolge in der Datei **App.config** entnehmen). Das Ergebnis sieht wie in Bild 3 aus.

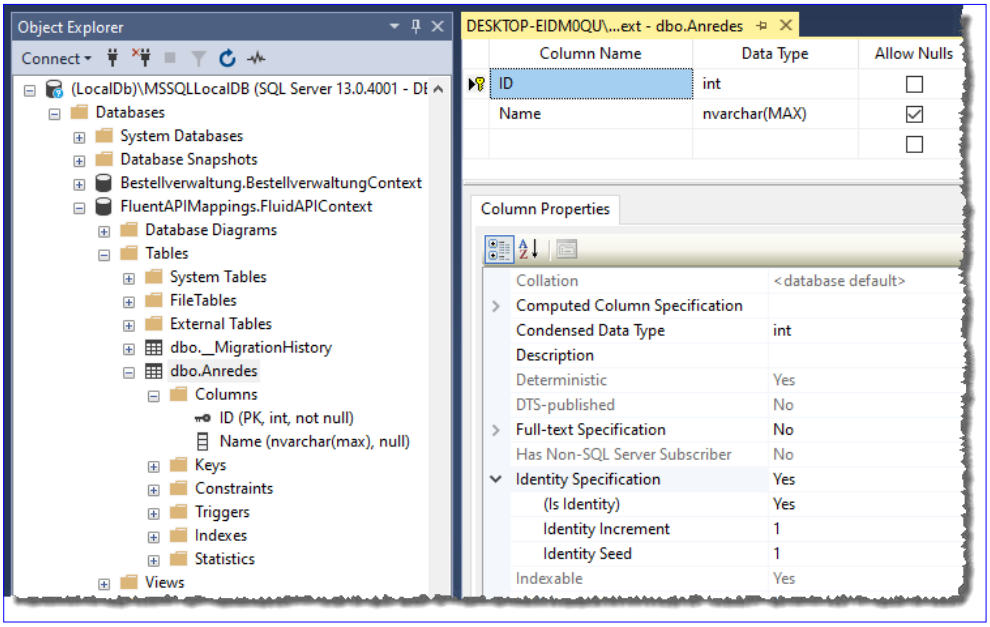


Bild 3: Die neue Tabelle in der frisch erzeugten Datenbank

Eingesetzte Konventionen

Hier erkennen wir gleich, dass bei der Erstellung der Tabelle einige Konventionen umgesetzt wurden. So hat Entity Framework automatisch erkannt, dass die Eigen-

schaft **ID** wohl als Primärschlüsselfeld mit Autowertfunktion verwendet werden soll. Außerdem hat es das Feld **Name** als Feld mit dem Datentyp **nvarchar(MAX)** angelegt, der Nullwerte erlaubt. Als Tabellenname hat Entity Framework aus der Entitätsklasse namens **Anrede** den Namen **Anredes** erzeugt. Dies resultiert ebenfalls aus einer Konvention, die dafür sorgt, dass Entity Framework den Tabellenamen aus dem Namen der Entitätsklasse plus dem **s** für den englischen Plural zusammenstellt.

Umgehen der Konventionen

In den folgenden Abschnitten schauen wir uns an, wie wir diese Konventionen durch Verwendung von Elementen der FluentAPI für das Entity Framework umgehen und mit eigenen Konventionen ersetzen können. Wir wollen zunächst die folgenden Dinge ändern:

- Der Tabellenname soll **Anreden** lauten oder gegebenenfalls auch **tblAnreden**.
- Der Felddatentyp des Feldes **Name** soll **nvarchar(50)** lauten.
- Das Primärschlüsselfeld soll den Namen **AnredeID** erhalten.
- Das Feld **Name** soll in der Tabelle den Namen **Anrede** erhalten.

Anpassen des Tabellennamens

Als Erstes setzen wir die Konvention außer Kraft, die dafür sorgt, dass die zu erstellende Tabelle den Namen der Entitätsklasse plus dem Plural-s als Bezeichnung erhält. Dazu fügen wir der Klasse **FluentApiContext** die Methode **OnModelCreating** hinzu und legen dort eine Anweisung an, welche die Konvention aufhebt:

```
Public Class FluidAPIContext
    Inherits DbContext
    ...
    Public Overridable Property Anreden As DbSet(Of Anrede)

    Protected Overrides Sub OnModelCreating(ByVal modelBuilder As DbModelBuilder)
        modelBuilder.Conventions.Remove(Of PluralizingTableNameConvention)()
    End Sub
End Class
```

Um den Typ **PluralizingTableNameConvention** zu nutzen, fügen wir einen weiteren Namespace zu der Klassendatei hinzu:

```
Imports System.Data.Entity.ModelConfiguration.Conventions
```

Nun aktualisieren wir das Datenmodell, indem wir die folgenden beiden Anweisungen in der Paket-Manager-Konsole aufrufen:

```
Add-Migration Init
Update-Database
```

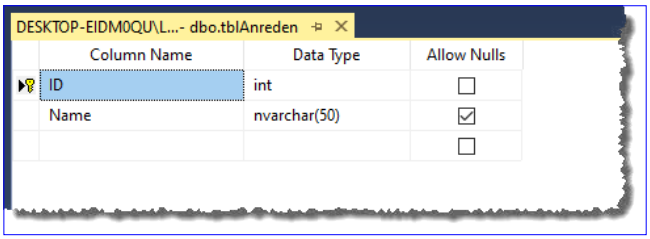


Bild 4: Angepasste Feldgröße

Danach finden wir die Tabelle in der Datenbank unter dem Namen **Anrede** vor. Wir möchten allerdings den Namen **Anreden** oder auch **tblAnreden**. Dazu fügen wir der Methode **OnModelCreating** eine weitere Anweisung hinzu:

```
modelBuilder.Entity(Of Anrede)().ToTable("Anreden")
```

Wenn Sie den Namen **tblAnreden** verwenden möchten, nutzen Sie einfach folgende Anweisung:

```
modelBuilder.Entity(Of Anrede)().ToTable("tblAnreden")
```

Um diese Änderungen durchzuführen, rufen Sie immer wieder die beiden Anweisungen in der Paket-Manager-Konsole auf – wir weisen in den folgenden Abschnitten nicht mehr explizit darauf hin. Außerdem müssen Sie, um die Änderungen in der SQL Server Management Konsole zu sehen, das Element **Tables** über den Kontextmenü-Eintrag **Refresh** aktualisieren.

Anpassen des String-Felddatentyps

Für das Feld Name der Entität **Anrede**, welche die Bezeichnung der Anrede aufnehmen soll, benötigen wir kein **nvarchar(MAX)**, **nvarchar(50)** reicht völlig aus. Zum Einstellen der Feldgröße fügen wir der Methode **OnModelCreating** den folgenden Befehl hinzu:

```
modelBuilder.Entity(Of Anrede)().Property(Function(t) t.Name).HasMaxLength(50)
```

Das sieht schon etwas komplizierter aus. Es geht um die Entität **Anrede** (**Entity (Of Anrede)**). Hier legen wir für die Eigenschaft **Name** über das Attribut **HasMaxLength(50)** die Länge **50** fest. Das Ergebnis sehen Sie in Bild 4.

Anpassen der Feldnamen

Nun wollen wir für das Primärschlüsselfeld den Namen **AnredeID** und für das Feld mit der Bezeichnung der Anrede den Namen **Anrede** statt **Name** verwenden. Hier treffen wir auf einen kleinen Konflikt, wenn wir die Tabelle beispielsweise sowohl mit einer Access-Datenbank als Frontendend verwenden wollen als auch in einem Visual Studio-Projekt. Unter Access kann es zu Problemen kommen, wenn wir **Name** als **Feldname** verwenden, unter Visual Studio darf eine Eigenschaft einer Entitätsklasse nicht so heißen wie die Klasse selbst. Also verwenden wir den Feldnamen **Anrede** und mappen diesen für das Visual Studio-Projekt auf die Eigenschaft **Name**. Damit das Feld **Name** in der Tabelle unter dem Namen **Anrede** erscheint, fügen wir die folgende Zeile zur Methode **OnModelCreating** hinzu:

```
modelBuilder.Entity(Of Anrede)().Property(Function(t) t.Name).HasColumnName("Anrede")
```

Das gleiche erledigen wir für die Eigenschaft **ID**, die in der Tabelle im Feld **AnredeID** erscheinen soll:

```
modelBuilder.Entity(Of Anrede)().Property(Function(t) t.ID).HasColumnName("AnredeID")
```

Das Ergebnis sieht nach der Änderung wie in Bild 5 aus.

Eigenschaft für das Primärschlüsselfeld festlegen

Wenn der von Ihnen gewählte Name für die Eigenschaft einer Entität, die als Primärschlüsselfeld dienen soll, nicht der Konvention entspricht und beispielsweise nicht die Zeichenfolge **ID** und den Datentyp **int** enthält, können Sie den Primärschlüssel explizit festlegen. Wir nehmen eine zweite Entitätsklasse namens **Person.vb** hinzu, die wie folgt aussieht:

```
Public Class Person
    Public Property PK As Integer
    Public Property Vorname As String
    Public Property Nachname As String
    Public Property Strasse As String
    Public Property PLZ As String
    Public Property Ort As String
    Public Property AnredeID As Integer
    Public Overridable Property Anrede As Anrede
End Class
```

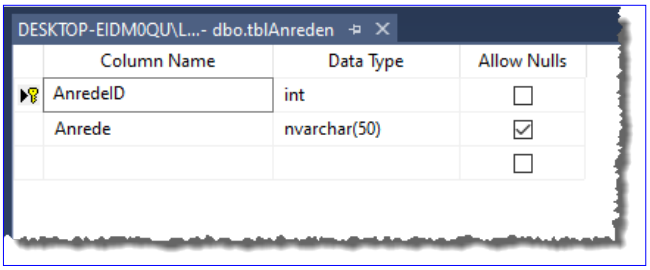


Bild 5: Angepasste Feldbezeichnungen

Wir haben hier absichtlich kein Primärschlüsselfeld namens **ID** oder **PersonID** hinzugefügt, sondern eines, dass keine Bezeichnung mit der Zeichenfolge **ID** enthält – daher heißt das Feld, das wir als Primärschlüsselfeld verwenden wollen, einfach **PK**.

Versuchen wir nun, in der Paket-Manager-Konsole mit **Add-Migration <Migrationsname>** eine neue Migration zu erstellen, erhalten wir eine Meldung, dass für die Entität **Person** kein Schlüssel definiert wurde. Dies erledigen wir explizit in der Methode **OnModelCreating**:

```
Protected Overrides Sub OnModelCreating(ByVal modelBuilder As DbModelBuilder)
    ...
    'Schlüssel für Tabelle festlegen
    modelBuilder.Entity(Of Person)().HasKey(Function(t) t.PK)
End Sub
```

Damit auch eine neue Tabelle für diese Klasse erstellt wird, fügen wir noch die folgende Zeile zur Klasse **FluidAPIContext** hinzu:

```
Public Overridable Property Personen As DbSet(Of Person)
```

Code First Mapping per DataAnnotation

Wenn Sie ein Code First-Entity Data Model entwerfen, gibt es bestimmte Konventionen, die standardmäßig greifen. So heißen Entitäten wie der Singular der zugrunde liegenden Tabellennamen oder der Primär- und Fremdschlüssel werden aus Feldern abgeleitet, welche die Zeichenkette ID enthalten – gegebenenfalls kombiniert mit dem Entitätsnamen. Wenn Sie mit einem Datenmodell daherkommen, dessen Tabellen das Präfix »tbl« mitbringen, wollen Sie dieses nicht in den Entitätsnamen wiederfinden und gegebenenfalls möchten (oder müssen) Sie auch noch Feldnamen ändern und diese anschließend mappen. Eine Möglichkeit dazu bieten die DataAnnotations, die wir in diesem Artikel beschreiben.

Im Artikel **Code First Mapping per Fluent API** (**www.datenbankentwickler.net/252**) haben wir gezeigt, wie Sie die Entitätsklassen eines Entity Data Models auf die Tabellen eines Datenmodells einer SQL Server-Datenbank mappen können beziehungsweise wie Sie dafür sorgen können, dass die Tabellen, Felder und Beziehungen der Tabellen beim Erstellen einer neuen Datenbank auf Basis von Code First nach Ihren Wünschen erstellt werden können. Dabei haben wir ausschließlich die Methoden der Fluent API verwendet, die in einer speziellen Methode namens **OnModelCreating** angelegt werden. Im vorliegenden Artikel wollen wir eine Alternative dazu vorstellen, nämlich die **DataAnnotations**. Dabei handelt es sich um »Anmerkungen«, die direkt in die jeweiligen Entitätsklassen eingetragen werden. Im Gegensatz zu den Methoden, die wir mit der Fluent API in der Methode **OnModelCreating** hinterlegen, landen die DataAnnotations unmittelbar bei den betroffenen Klassen und Eigenschaften. Wir wollen versuchen, die Mapping-Eigenschaften, die wir im oben genannten Artikel mit der Fluent API vorgenommen haben, mit den DataAnnotations zu reproduzieren. Wir verwenden also die gleichen Entitätsklassen und wollen das gleiche Datenmodell daraus erstellen beziehungsweise mappen wie in diesem Artikel.

Erstellen versus Mapping

In den Beispielen dieses Artikels, in denen wir die Anpassungen von Entity Data Model zu den Tabellen des zu erzeugenden Datenmodells Schritt für Schritt anpassen, erstellen wir die Datenbank einmal neu und führen die nachfolgend beschriebenen Änderungen Schritt für Schritt aus. Das erledigen wir mit drei Methoden, die wir über die Paket-Manager-Konsole absetzen. Diese Methoden lauten:

- **Enable-Migration**: Aktiviert die Migrationen für das Entity Data Model. Dieser Schritt ist nur einmal vor dem ersten Aufruf der nachfolgend beschriebenen Anweisung **Add-Migration** notwendig.

- **Add-Migration <Bezeichnung>**: Fügt eine neue Migration hinzu. Beim ersten Aufruf wird dabei eine Klasse erstellt, die Methoden mit allen Befehlen enthält, um die Datenbank und die enthaltenen Tabellen samt Feldern, Indizes und Beziehungen zu erstellen. Bei Aufrufen nach Änderungen am Entity Data Modell werden zu der jeweils neu erstellten Klasse nur Methoden mit Befehlen hinzugefügt, die für das Anwenden der Änderungen notwendig sind. Die Klassen und die enthaltenen Methoden und Befehle werden durch die nachfolgend vorgestellte Anweisung angewendet.

- **Update-Database**: Erstellt die Datenbank beziehungsweise nimmt die Änderungen vor, die durch **Add-Migration** hinzugefügt wurden.

Beispielprojekt für DataAnnotations

Das Beispielprojekt setzen wir als WPF-App für das .NET Framework mit Visual Basic auf. Diesem fügen wir ein neues Element des Typs **ADO.NET Entity Data Model** namens **DataAnnotationContext** hinzu. Im Dialog **Modellinhalt auswählen** selektieren wir **Leeres Code First-Modell**. Dies fügt die Klasse **DataAnnotationContext.vb**, einige Verweise und Informationen über das Entity Framework und die Verbindungszeichenfolge zur Datei **App.config** hinzu. Normalerweise beziehen sich diese auf eine Datenbank, die den gleichen Namen wie das Projekt plus einem Punkt plus dem Namen der Context-Klasse hat und die mit der aktuellen SQL Server LocalDb-Instanz verwaltet wird.

Entitätsklassen hinzufügen

Zu Beispielzwecken benötigen wir die gleichen Klassen wie im oben genannten Artikel. Diese heißen **Person.vb** und **Anrede.vb** sowie **Fahrzeug.vb** und **Ausstattungsmerkmal.vb**. Wie diese aussehen, entnehmen Sie dem Beispielprojekt, eine Beschreibung finden Sie im oben genannten Artikel. Wichtig ist: Zwischen **Person.vb** und **Anrede.vb** gibt es eine 1:n-Beziehung, zwischen **Fahrzeug.vb** und **Ausstattungsmerkmal.vb** eine m:n-Beziehung.

DbSet-Elemente hinzufügen

Damit der Aufruf der Anweisung **Add-Migration** überhaupt Anweisungen zum Erstellen von Tabellen in eine neue Klasse schreibt, benötigt sie Informationen darüber, welche Tabellen erstellt werden sollen. Diese Information hinterlegen wir in Form von **DbSet**-Deklarationen in der Klasse **DataAnnotationContext.vb**:

```
Public Overridable Property Anreden As DbSet(Of Anrede)
Public Overridable Property Personen As DbSet(Of Person)
Public Overridable Property Fahrzeuge As DbSet(Of Fahrzeug)
Public Overridable Property Ausstattungsmerkmale As DbSet(Of Ausstattungsmerkmal)
```

Erste Migration

Damit können wir die erste Migration durchführen und rufen die Befehle **Enable-Migrations** (erstellt den Ordner **Migrations** mit der Datei **Configuration.vb**) und **Add-Migration init** auf. Anschließend erstellen wir mit **Update-Database** eine Datenbank auf Basis der in der soeben erstellten Datei, deren Dateiname auf **_init.vb** endet. Wenn Sie mit dem Menübefehl **Ansicht|SQL Server-Objekt-Explorer** den Bereich **SQL Server-Objekt-Explorer** aktivieren, können Sie hier eine Verbindung mit der in der Verbindungszeichenfolge angegebenen SQL Server-Instanz herstellen und sich die Datenbank mit ihren Tabellen anzeigen lassen. Diese zeigt nicht nur die vier Tabellen für die von uns hinzugefügten Entitätsklassen an, sondern auch noch eine Verknüpfungstabelle zum Herstellen der Beziehung zwischen den beiden Klassen **Fahrzeug** und **Ausstattungsmerkmal**.

Datenbank neu erstellen

Wenn Sie die Datenbank neu erstellen wollen, sind vor dem Migrieren zwei Schritte nötig:

Auf die gleiche Weise passen wir die Namen der übrigen Tabellen an. Mit diesem Schritt haben wir auch bereits das Problem behoben, dass Entity Framework die Entitätsklasse **Person** in eine Tabelle namens **People** mappt. Beachten Sie, dass die **DataAnnotation**-Anweisungen immer unmittelbar über der Zeile mit der betroffenen Klasse angegeben werden müssen. Sollten Sie mehrere **DataAnnotation**-Anweisungen benötigen, tragen Sie diese Zeilen direkt untereinander über der ersten Zeile der Klassendefinition ein.

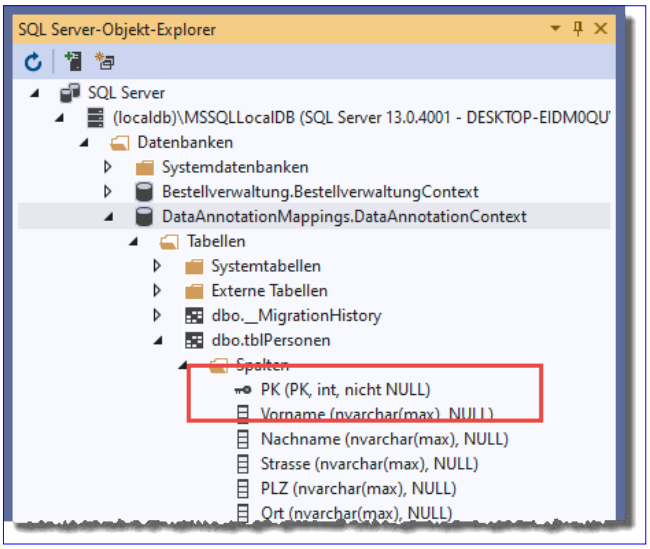


Bild 2: Das Feld **PK** als Primärschlüsselfeld

Primärschlüsselfelder anpassen

Um eine Primärschlüsseleigenschaft, die keine Bezeichnung aufweist, die Entity Framework automatisch als Primärschlüsselbezeichnung erkennt, dennoch als Primärschlüsselfeld zu verwenden, nutzen Sie das **DataAnnotation**-Element **Key**.

Dieses fügen wir wie folgt hinzu – hier im Falle des Feldes **PK**:

```
<Table("tblPersonen")>
Public Class Person
    <Key>
    Public Property PK As Integer
```

Damit erhalten wir in der Tabelle ein Primärschlüsselfeld namens **PK** (siehe Bild 2). Außerdem wollen wir aber dafür sorgen, dass das Primärschlüsselfeld die Bezeichnung **PersonID** erhält. Dazu fügen wir zusätzlich zu **Key** das **DataAnnotation**-Element **Column** hinzu:

```
<Key>
<Column("PersonID")>
Public Property PK As Integer
```

Dies ändert den Namen des Primärschlüsselfeldes der Tabelle **tblPersonen** von **PK** in **AnredeID**. Auf die gleiche Weise passen wir auch noch die Feldnamen der übrigen Entitätsklassen an. Benötigen wir überall die **Key**-**DataAnnotation**? Nein, eigentlich nicht, denn Entity Framework erkennt die übrigen Primärschlüsseleigenschaften am Namen **ID**. Es kann aber auch nicht schaden, Entity Framework hier fest vorzugeben, welche Eigenschaft als Primärschlüsselfeld in der Tabelle landen soll.

Bezeichnungen weiterer Felder anpassen

Für die Entitätsklasse **tblAnreden** wollen wir das Feld **Name** im Tabellenentwurf in **Anrede** umwandeln. Dazu stellen wir auch diesem Feld die passende **Column**-**DataAnnotation** voran:

```
<Column("Anrede")>
Public Property Name As String
```

Datentyp von Tabellenfeldern einstellen

Wenn wir den Datentyp eines Feldes ändern wollen, stellen wir diesen ebenfalls mit dem **DataAnnotation**-Element **Column** ein, und zwar mit dem benannten Parameter **TypeName**:

```
<Column("Anrede", TypeName:="String")>
Public Property Name As String
```

Wenn Sie zusätzlich noch die Feldgröße anpassen möchten, gelingt das mit dem zusätzlichen **DataAnnotation**-Element **StringLength**, dem Sie als Parameter die gewünschte Feldgröße übergeben:

```
<Column("Anrede", TypeName:="nvarchar")>
<StringLength(50)>
Public Property Name As
String
```

Damit erhalten wir ein Feld namens **Anrede** mit dem Typ **nvarchar(50)**.

Primärschlüsselfeld nicht automatisch als Autowert-Feld definieren

Nicht immer ist es gewünscht, dass ein Primärschlüsselfeld in einer Tabelle beim Anlegen neuer Datensätze automatisch mit inkrementierten Werten gefüllt wird. Gegebenenfalls möchten Sie die Primärschlüsselwerte selbst festlegen. Damit dies beim Erstellen des Datenmodells auf Basis von Code First Berücksichtigung findet, fügen Sie

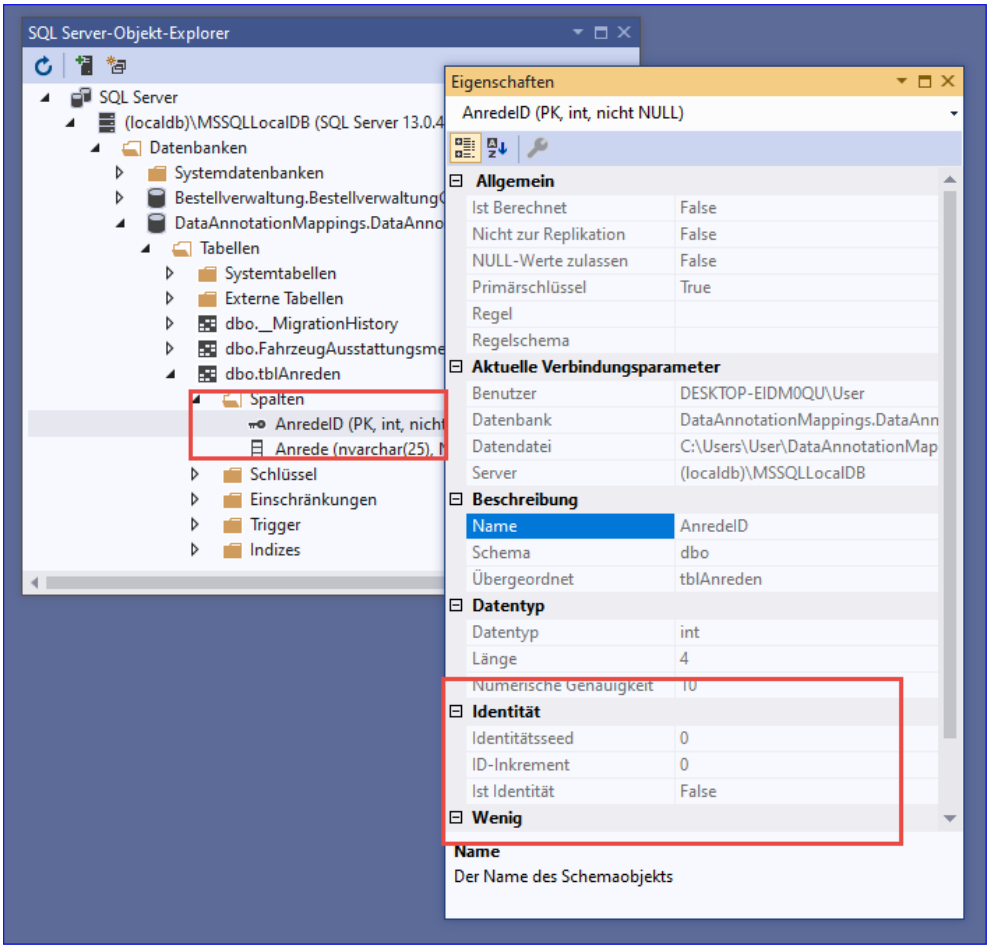


Bild 3: Steuern der Identitätswerte des Primärschlüsselfeldes einer Tabelle

das **DataAnnotation**-Element **DatabaseGenerated** mit dem gewünschten Parameterwert hinzu. Die interessanten Werte lauten:

- **DatabaseGeneratedOption.None**: Keine automatische Erstellung von Primärschlüsselwerten. Das Primärschlüsselfeld muss dann vor dem Speichern der Daten mit SaveChanges per Code gesetzt werden.

- **DatabaseGeneratedOption.Identity**: Ermittelt den neuen Primärschlüsselwert wie normalerweise üblich.

Da Entity Framework die letztere Option per Konvention verwendet, geben wir in diesem Fall den Wert **DatabaseGeneratedOption.None** an:

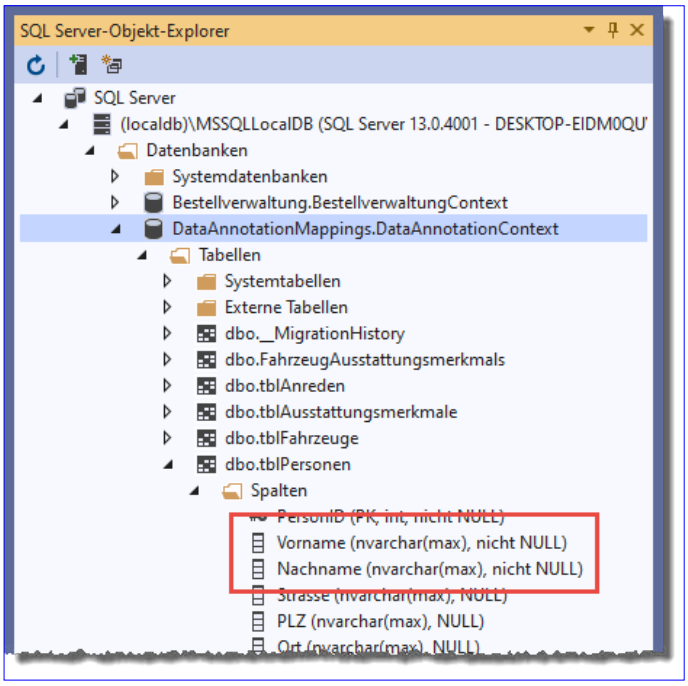


Bild 4: Die beiden Felder dürfen nicht leer sein.

```
<DatabaseGenerated(DatabaseGeneratedOption.None)>
 Public Property ID As Integer
```

Das führt zu dem Ergebnis aus Bild 3.

Keine Nullwerte für bestimmte Felder

Wenn Sie beispielsweise festlegen wollen, dass die beiden Felder **Vorname** und **Nachname** der Tabelle **tblPersonen** keine Nullwerte enthalten dürfen, verwenden Sie das **DataAnnotation**-Element **Required** für die jeweiligen Felder:

```
<Table("tblPersonen")>
Public Class Person
    <Key>
    <Column("PersonID")>
    Public Property PK As Integer
    <Required>
    Public Property Vorname As String
    <Required>
    Public Property Nachname As String
    ...
End Class
```

Das Ergebnis nach der Migration sehen Sie in Bild 4.

SQL Server Reporting Services

Wer von Access kommt, kennt die dort verfügbare Darstellung von Daten in Berichtsform. Berichte lassen sich mit wenigen Kenntnissen zusammenklicken und werden dann Teil der Datenbankdatei. Wenn Sie in Mehrbenutzerumgebungen Daten als Bericht anzeigen wollen und dabei regelmäßig neue Berichte erstellen, wird das recht aufwändig – neue Berichte müssen dann immer in Form eines neuen Frontends verteilt werden. Oder Sie erstellen ein eigenes Frontend nur für die Berichte. Wie auch immer: Wenn Sie .NET-Anwendungen programmieren, stehen die Access-Berichte nicht mehr bereit. Eine Alternative lautet SQL Server Reporting Services. Diese bieten einen anderen Ansatz als die Access-Berichte: Sie stellen ihre Daten über eine Service bereit, den Sie über den Browser nutzen können. Der Benutzer benötigt also noch nicht einmal Zugriff auf eine Datenbankanwendung, um Berichtsdaten einzusehen. Der vorliegende Artikel liefert grundlegende Informationen über die SQL Server Reporting Services.

Die SQL Server Reporting Services

Bei den SQL Server Reporting Services handelt es sich um ein eigenständiges Produkt, dass aber eng mit den Programmiertools verknüpft ist. Seine Aufgabe ist es nicht nur, wie Access Berichte auf Basis der in den Tabellen gespeicherten Daten nach verschiedenen Kriterien aufzubereiten und auszugeben.

Es ist eher eine komplette Infrastruktur zur Erstellung, Verwaltung und Bereitstellung von Berichten in einem Unternehmen. Damit erhalten Sie mit den SQL Server Reporting Services einen Baustein für den Bereich der Business Intelligence. Damit soll es möglich sein, die für verschiedene Bereiche des Unternehmens notwendigen Daten zu liefern – ob übersichtliche Dashboards für die Geschäftsführung, detaillierte Daten für das Controlling oder einfach Artikellisten mit Verfügbarkeiten und Lieferzeiten.

Installation und Co.

Wie Sie die SQL Server Reporting Services, kurz SSRS, installieren, welche weiteren Elemente Sie dafür benötigen und wie Sie erste Beispielberichte damit erstellen und veröffentlichen, zeigen wir beispielsweise in den Artikeln **Reporting Services 2019 installieren und starten** (**www.datenbankentwickler.net/256**) und **Reporting Services: Web-Portal** (**www.datenbankentwickler.net/257**).

Der vorliegende Artikel soll grundlegende Informationen zu den SQL Server Reporting Services liefern.

Was können die SQL Server Reporting Services?

Mit den SSRS können Sie Berichte erzeugen, bereitstellen und verwalten. Dabei erfolgt der Zugriff durch den Anwender (nicht zu verwechseln mit dem Programmierer der Berichte) über eine Webschnittstelle. Mit den SSRS können Sie Berichte in verschiedenen Formaten erzeugen, zum Beispiel:

- Paginierte Berichte in Form von PDF-Dateien, Excel-Dateien, Word-Dateien et cetera

- Mobile Berichte, die der Benutzer auf Smartphones oder Tablet lesen kann. Diese Berichte werden speziell darauf ausgelegt, auf unterschiedlichen Bildschirmgrößen dargestellt zu werden.

- Webbasierte Berichte für die Anzeige im Webbrowser

Berichte enthalten Elemente wie Texte, Tabellen, Grafiken, Bilder oder Diagramme.

Berichte definieren mit der Report Definition Language (RDL)

Die Sprache für die Definition von Berichten heißt **Report Definition Language**, kurz **RDL**. Dabei handelt es sich um eine XML-basierte Sprache. Ein RDL-Dokument enthält die kompletten Informationen, die zum Erstellen eines Berichts notwendig sind. Diese teilen sich auf in Informationen für das Ermitteln der Daten und das Layout, in dem die Daten dargestellt werden sollen. Durch das XML-Format ist auch ein Austausch mit anderen Anwendungen möglich.

Die RDL-Datei als das Ergebnis des Designs eines Berichts enthält somit Vorgaben dafür, wie die aus den angegebenen Datenquellen ermittelten Daten angeordnet werden sollen. Details über die Sprache RDL benötigen Sie in der Regel nicht, denn es gibt verschiedene Tools, mit denen Sie Berichte per Drag and Drop über eine entsprechende Benutzeroberfläche zusammenstellen können.

Microsoft SQL Server Services für Business Intelligence

Die SQL Server Reporting Services sind nicht allein, wenn es darum geht, Business Intelligence-Auswertungen bereitzustellen. Es gibt noch zwei weitere Services neben den SSRS, die für den Datenzugriff und die Auswertung verwendet werden können. Insgesamt haben wir also:

- **SQL Server Reporting Services** (**SSRS**)

- **SQL Server Analysis Services** (**SSAS**)

- **SQL Server Integration Services** (**SSID**)

SQL Server Analysis Services (SSAS)

Die SQL Server Analysis Services, kurz SSAS, dienen der Analyse umfangreicher Datenmengen mit verschiedenen Methoden und Verfahren. Damit erhalten Sie ein Tool, das multidimensionales Online Analytical Processing und Data Mining im Business Intelligence-Bereich erlaubt. Die zu analysierenden Daten können aus verschiedenen Datenquellen wie relationalen Datenbanken, XML-Dokumenten oder auch Textdateien stammen und werden mit den nachfolgend vorgestellten SQL Server Integration Services erfasst. Die SSAS erlauben ein besseres Verständnis von Geschäftsdaten eines Unternehmens.

SQL Server Integration Services (SSIS)

Die SQL Server Integration Services, kurz SSIS, haben die Aufgabe, Daten aus unterschiedlichen Datenquellen zusammenzuführen. Mit den SQL Server Integration Services fassen Sie Daten aus verschiedenen Quellen

zusammen. Die Art des Datenformats ist dabei recht flexibel und reicht von relationalen Datenbanken über XML-Dateien, CSV-Dateien bis hin zu Excel-Dateien. Die SSIS extrahieren, laden, transformieren, bereinigen, aggregieren oder kopieren Daten oder führen diese zusammen. Die von den SQL Server Integration Services gelieferten Daten werden mit Tools wie SQL Server Analysis Services oder SQL Server Reporting Services weiterverarbeitet.

Tools zum Erstellen von Berichten

Um die von den SQL Server Reporting Services gelieferten Berichte zu erstellen, stellt Microsoft verschiedene Tools zur Verfügung. Eines von beiden verwenden wir in den Artikeln **Reporting Services: Web-Portal** (**www.datenbankentwickler.net/257)** oder **Reporting Services: Entwurf von Berichten** (**www.datenbankentwickler.net/202**). Dabei handelt es sich um Visual Studio mit entsprechenden Erweiterungen.

Außerdem können Sie den Report Builder von Microsoft verwenden. Diesen können Sie direkt bei Microsoft herunterladen – suchen Sie einfach bei Google nach **SQL Server Report Builder Download**. Wie Sie diesen nutzen, beschreiben wir im Artikel **SQL Server Report Builder** (**www.datenbankentwickler.net/258)**.

Zum Erstellen von Berichten für mobile Endgeräte benötigen Sie den **SQL Server Mobile Report Publisher**.

Das Web-Portal

Reporting Services bieten ein modernes Web-Portal, mit dem Sie die verschiedenen Typen von Berichten anzeigen können – paginierte Berichte, mobile Berichte, KPIs oder Excel-Workbooks. Das Web-Portal ersetzt den zuvor angebotenen **Report Manager**.

Wenn die Reporting Services laufen und Sie die URL kennen, können Sie das Web-Portal einfach im Webbrowser öffnen. Wenn die Reporting Services auf dem lokalen Rechner gestartet wurden, verwenden Sie dazu die folgende Adresse:

```
http://localhost/Reports/
```

Einige Grundlagen über das Web-Portal liefert der Artikel **Reporting Services: Web-Portal** (**www.datenbankentwickler.net/257**).

SharePoint

Die Berichte können auch in ein Berichtsportal innerhalb von SharePoint integriert werden. SharePoint ist jedoch ein ganz anderes Thema und wir werden darauf an dieser Stelle nicht eingehen. Wichtig ist es hier zu wissen, dass Sie bei der Installation für die Verwendung mit SharePoint nicht den nativen Modus wählen, sondern die SharePoint-Integration.

Der Berichtsserver

Als Berichtsserver bezeichnen wir den Computer, auf dem die Instanz der Reporting Services läuft. Er speichert die paginierten oder mobilen Berichte und andere Informationen rund um das Reporting. Diese können in

Reporting Services 2019 installieren und starten

Der Vorteil von Microsoft Access ist: Man bekommt alles aus einer Hand. Tabellenentwurf, Abfragedesigner, Formulare, Programmierumgebung und – Tools zur Berichterstellung. Bei der Datenbank mit Visual Studio gibt es so viele Möglichkeiten, die aber alle irgendwie viel größer und komplizierter erscheinen als das, was der Berichtsdesigner von Access bietet. Früher oder später wollen wir aber auch in DATENBANKENTWICKLER das Thema Reporting behandeln und deshalb schauen wir uns in dieser Artikelreihe die Möglichkeiten der SQL Server Reporting Services an. Da SQL Server mittlerweile in einer Community Version kommt, die für den privaten Einsatz kostenlos ist, steht dem Ausprobieren auch nichts im Wege. Der vorliegende Artikel ist eine Neuauflage des Artikels »Reporting Services: Installation und Start« (www.datenbankentwickler.net/200). Diese ist notwendig, weil sich einige Schritte beim Installieren geändert haben.

Schritt für Schritt

In den folgenden Abschnitten zeigen wir Ihnen, welche Tools und Software Sie benötigen, um die Reporting Services für den SQL Server in der Version von 2019 verwenden zu können. Dabei erledigen wir die folgenden Schritte:

- Installation der **SQL Server 2019 Reporting Services**
- SQL Server-Instanz für die Verwendung von **SQL Server 2019 Reporting Services** festlegen
- **SQL Server Data Tools** installieren
- **SQL Server Integration Services Projects** installieren
- **Microsoft Analysis Services Projects** installieren
- **Microsoft Reporting Services Projects** installieren

Bild 1: Installation von **SQL Server 2019 Reporting Services**

Installation der Reporting Services

Die SQL Server Reporting Services finden Sie, wenn Sie bei Google nach den Begriffen **Installieren von SQL Server Reporting Services** suchen. Auf der Seite von Microsoft mit diesem Titel finden Sie einen Downloadlink für **SQL Server 2019 Reporting Services**.

Die dort heruntergeladene Datei **SQLServerReportingServices.exe** starten Sie anschließend, um die **SQL Server 2019 Reporting Services** zu installieren. Hier finden Sie zunächst nur die Option aus Bild 1 vor.

Im folgenden Schritt wählen Sie dann aus, welche Edition Sie installieren wollen. Wenn Sie eine gekaufte Lizenz haben, verwenden Sie natürlich den entsprechenden **Product Key**.

Anderenfalls erhalten Sie mit der Entwickler-Edition, die wir in Bild 2 ausgewählt haben, den vollen Funktionsumfang. Ein Klick auf **Weiter** zeigt die Lizenzvereinbarungen an.

Danach folgen die Abfrage, ob Sie bereits einen SQL Server installiert haben, was Sie gegebenenfalls nun nachholen können. Danach wählen Sie den Installationsspeicherort aus, wo Sie in der Regel das vorgegebene Verzeichnis übernehmen können, und starten die Installation mit einem Klick auf die Schaltfläche **Installieren** (siehe Bild 3).

Nach wenigen Augenblicken hat der Installationsassistent **SQL Server 2019 Reporting Services** installiert und zeigt den Abschlussbildschirm an (siehe Bild 4).

Hier haben Sie nun die Möglichkeit, den Berichtsserver manuell zu konfigurieren. Dazu klicken Sie auf die Schaltfläche **Berichtsserver konfigurieren**.

Bild 2: Zweiter Schritt des Assistenten

Bild 3: Start der Installation

Bild 4: Abschluss und weitere Optionen

SQL Server-Instanz auswählen

Dies startet den **Berichtsserver-Konfigurations-Manager**. Dieser verlangt zunächst nach der Auswahl des zu verwendenden SQL Servers. Hier wollen wir, da wir den Berichtsserver für die Beispieldatenbanken einer **LocalDb**-Instanz nutzen wollen, diese Instanz wählen. Allerdings gelingt dies nicht: **LocalDb** ist eine Instanz, die in der Desktop Session läuft, und daher nicht für den Einsatz mit den **SQL Server 2019 Reporting Services** zur Verfügung steht. Daher geben Sie eine als Service laufende SQL Server-Instanz an und bestätigen den Dialog mit **Verbinden** (siehe Bild 5).

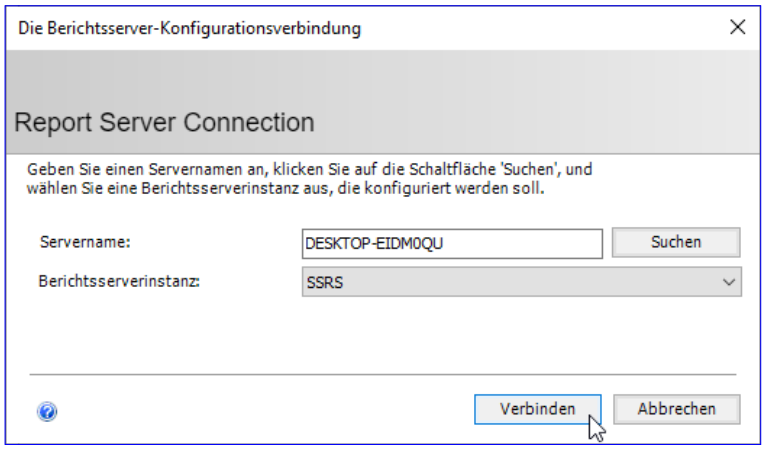


Bild 5: Einstellen der Verbindung für den Report Server

Report Server nachträglich anpassen

Wenn Sie diesen Schritt auslassen oder die Konfiguration nachträglich ändern wollen, können Sie den Berichtsserver-Konfigurations-Manager auch später noch starten. Dazu geben Sie im Suchfeld von Windows den Suchbegriff **Report Server Configuration Manager** ein und starten diesen Eintrag anschließend. Die Einstellungen zeigt der Berichtsserver-Konfigurations-Manager anschließend in einer Übersichtsseite an, die Sie auch erhalten, wenn Sie diesen erneut starten (siehe Bild 6).

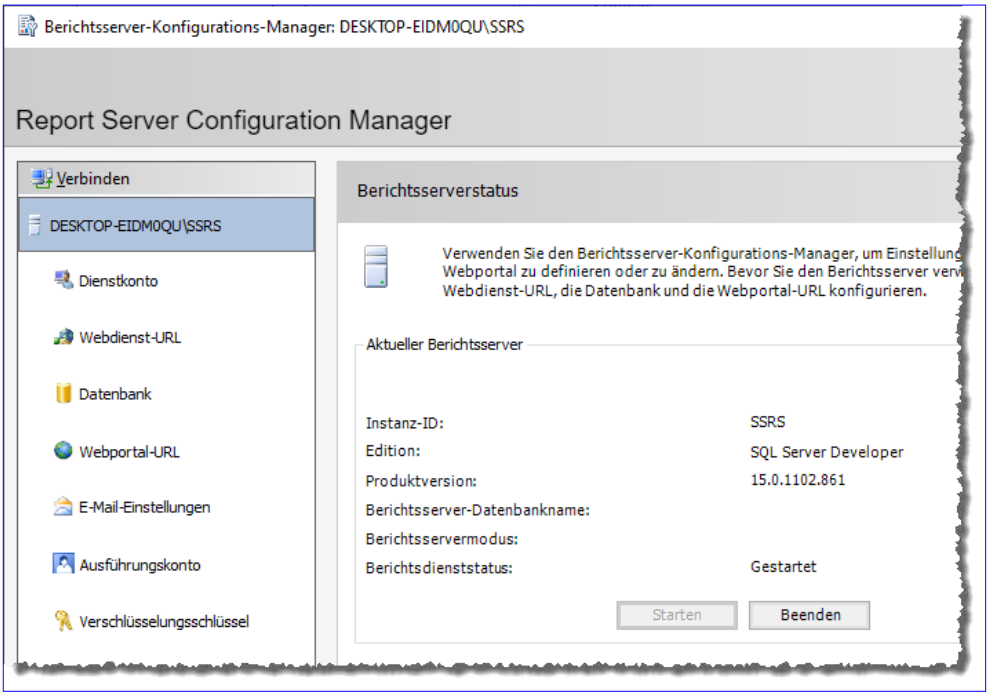


Bild 6: Der Berichtsserver-Konfigurations-Manager

Report Server-Datenbank anlegen

Hier sind noch weitere Einstellungen nötig. Zum Beispiel legen wir eine Datenbank an, welche die Reporting Services nutzen können. Dazu wählen Sie im linken Bereich des **Report Server Configuration Manager** den Eintrag **Datenbank** aus. Anschließend klicken Sie im nun erscheinenden Bereich **Berichtsserver-Datenbank** auf die Schaltfläche **Datenbank ändern** (siehe Bild 7).

Bild 7: Einstellen einer Datenbank für die Reporting Services

Im nun erscheinenden Dialog **Assistent zum Konfigurieren der Berichtsserver-Datenbank** behalten Sie die Option **Neue Berichtsserver-Datenbank erstellen** bei und klicken auf die Schaltfläche **Weiter** (siehe Bild 8).

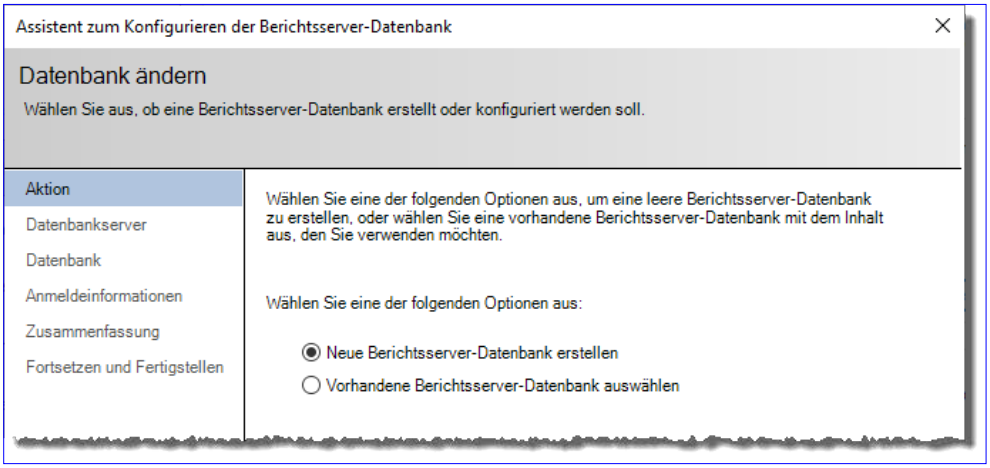


Bild 8: Erstellen einer neuen Datenbank für die Reporting Services

Im nächsten Schritt namens **Datenbankserver** behalten Sie die aktuellen SQL Server-Instanz bei oder wählen eine andere Instanz aus. Der folgende Schritt mit der Bezeichnung **Datenbank** erwartet die Angabe eines Namens für die zu erstellende Datenbank. Hier behalten wir

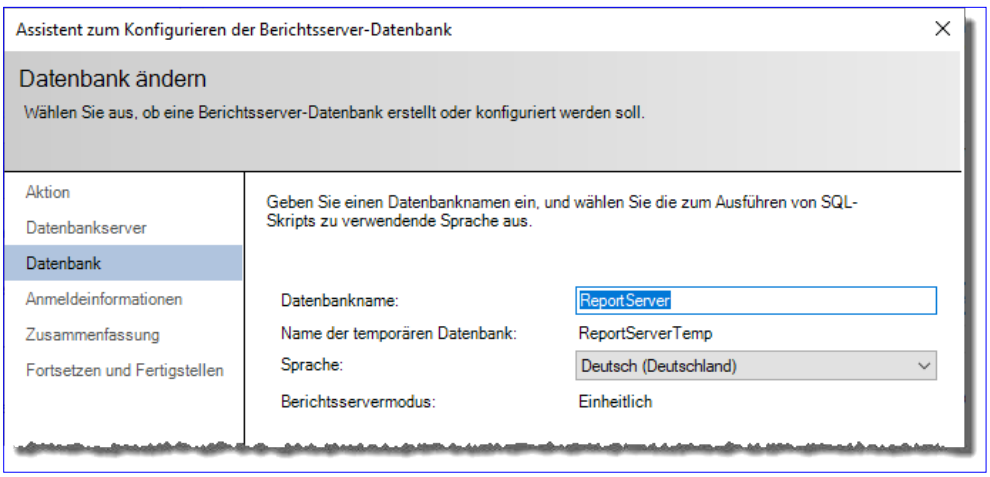


Bild 9: Anlegen der neuen Datenbank für die Reporting Services

den Wert **ReportServer** bei (siehe Bild 9). Die übrigen Voreinstellungen des Assistenten können Sie beibehalten und diesen mit mehreren Klicks auf die jeweilige **Weiter**-Schaltfläche und im letzten Schritt auf die Schaltfläche **Fertig stellen** beenden.

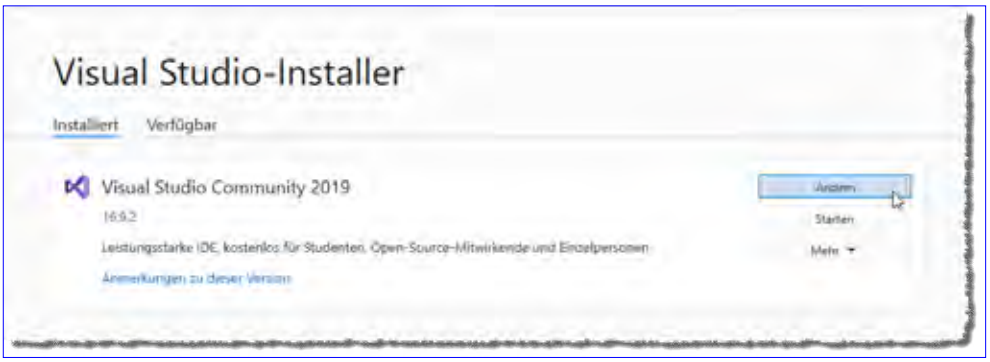


Bild 10: Der Visual Studio-Installer

Webdienst-URL und Webportal-URL

Auf den beiden Seiten **Webdienst-URL** und **Webportal-URL** behalten Sie die Einstellungen bei und klicken jeweils auf die Schaltfläche **Anwenden**. Die URL unter Webportal-URL benötigen Sie später noch; hier finden Sie die Berichte vor.

Installation der SQL Server Data Tools

Eine umfangreiche Neuerung betrifft die Installation der SQL Server Data Tools. In früheren Versionen wurden diese noch als eigenes Setup installiert. Das hat sich geändert – die Tools gehören nun zum Setup von Visual Studio. Also starten Sie über das Suchfeld von Windows das Programm **Visual Studio Installer**.

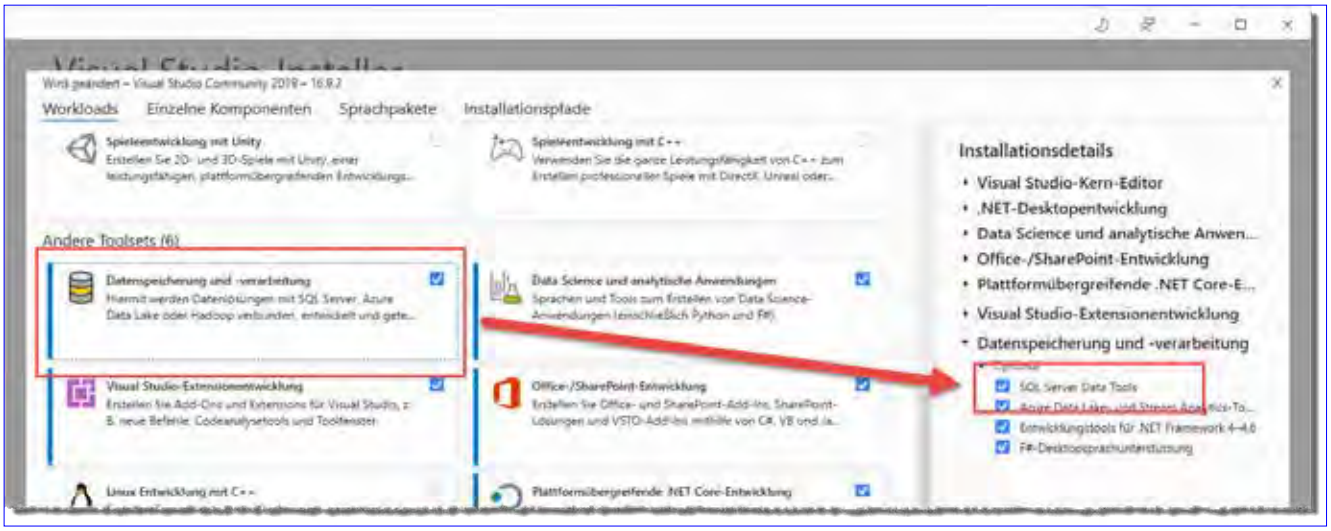


Bild 11: Aktivieren der SQL Server Data Tools

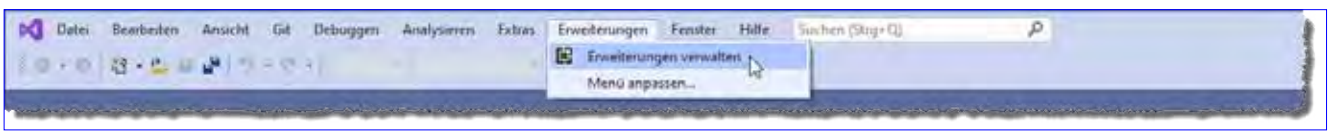


Bild 12: Aufruf der Verwaltung der Erweiterungen

SQL Server Report Builder

Neben Visual Studio gibt es noch ein weiteres, wesentlich schlankeres Tool zum Definieren von Berichten auf Basis der SQL Server Reporting Services. Dieses Tool heißt Report Builder und kann kostenlos bei Microsoft heruntergeladen werden. Es bietet sich als Alternative für solche Anwendungszwecke an, wo nur Berichte designed werden sollen und die übrigen Funktionen von Visual Studio nicht erforderlich sind. Dieser Artikel zeigt, wie Sie den Report Builder installieren und verwenden.

Download des Report Builders

Den Download des Report Builders finden Sie, wenn Sie in einer Suchmaschine den Begriff **SQL Server Report Builder** eingeben.

Installation des Report Builders

Die Installation erfolgt auf einfache Weise durch Starten der zuvor heruntergeladenen **.msi**-Datei. Der einzige Schritt, in dem Sie etwas erledigen können, ist der aus Bild 1.

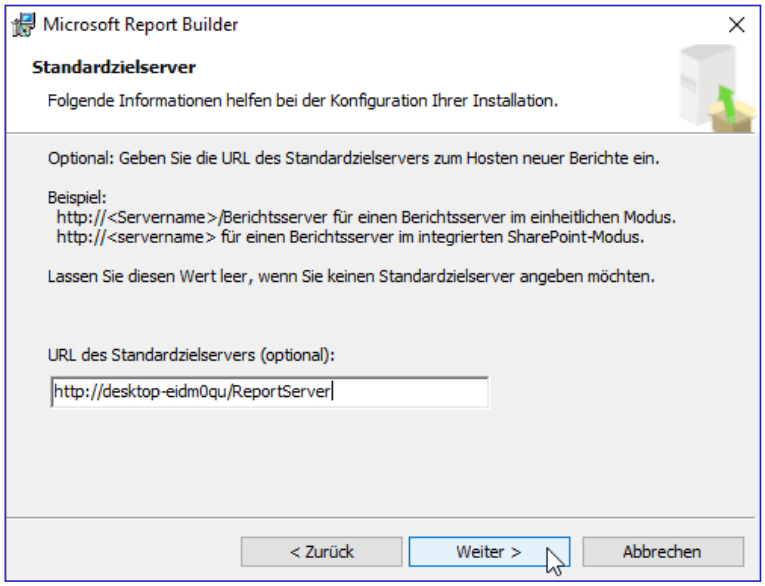


Bild 1: Angabe des Berichtsservers während der Installation

Für die Eigenschaft **URL des Standardzielservers** geben Sie die im Report Server Configuration Manager angegebene URL unter **Webdienst-URL** an. Den Report Server Configuration Manager öffnen Sie über das **Suchen**-Textfeld von Windows.

Im **Report Server Configuration Manager** wechseln Sie zum Bereich **Webdienst-URL** und ermitteln dort unter URLs für Berichtsserver-Webdienst die gesuchte URL (siehe Bild 2).

In unserem Fall lautet die hier einzutragende URL:

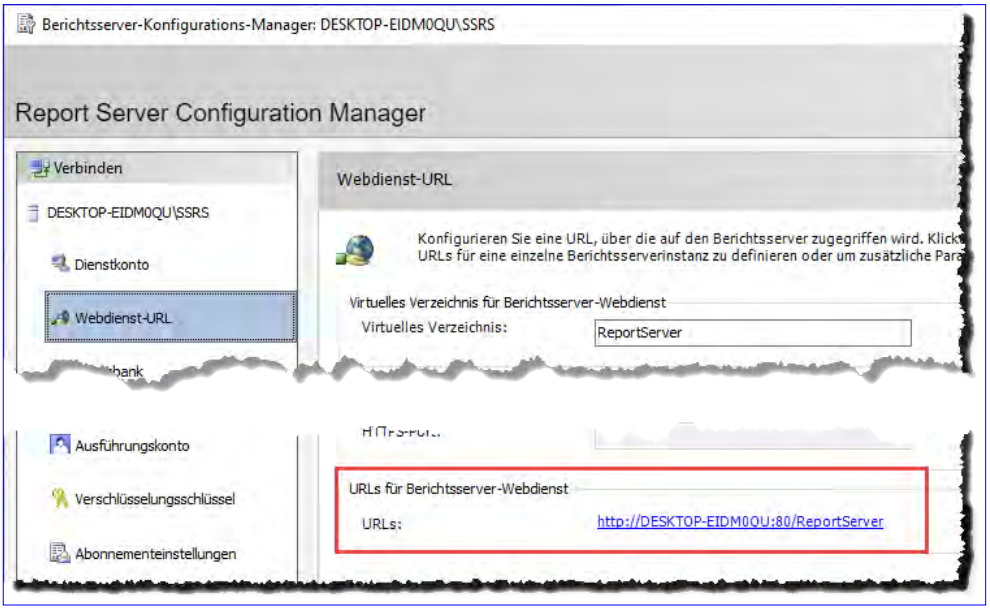


Bild 2: Ermitteln der im Report Server Configuration Manager definierten Webdienst-URL

```
http://desktop-eidm0qu/ReportServer
```

Wenn die Reporting Services auf dem lokalen Rechner betrieben werden, können Sie auch **localhost** als Server angeben.

Danach starten Sie die Installation, die innerhalb weniger Sekunden abgeschlossen ist. Den Report Builder können Sie anschließend über das Suchen-Fenster von Windows finden und starten.

Report Builder vom Web-Portal aus starten

Eine weitere Möglichkeit, den Report Builder zu starten, ist das Web-Portal. Dieses finden Sie im Webbrowser unter **<Servername>/Reports**, für Reporting Services auf dem lokalen Rechner also beispielsweise **localhost/Reports**. Wenn Sie hier auf das Plus-Zeichen klicken, erscheint eine Liste von Optionen zum Anlegen neuer Elemente (siehe Bild 3).

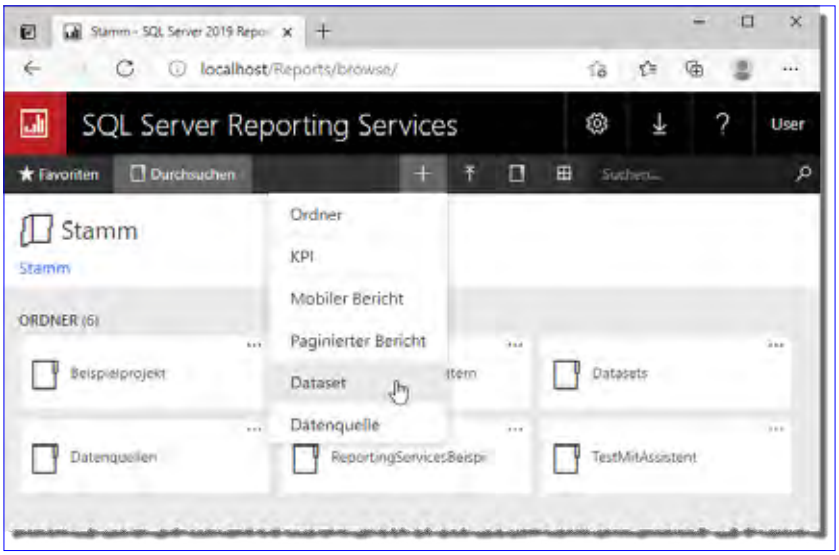


Bild 3: Anlegen neuer Objekte über das Web-Portal

Wenn Sie den Report Builder zuvor noch nicht vom Web-Portal aus geöffnet haben, erscheinen die Meldungen aus Bild 4.

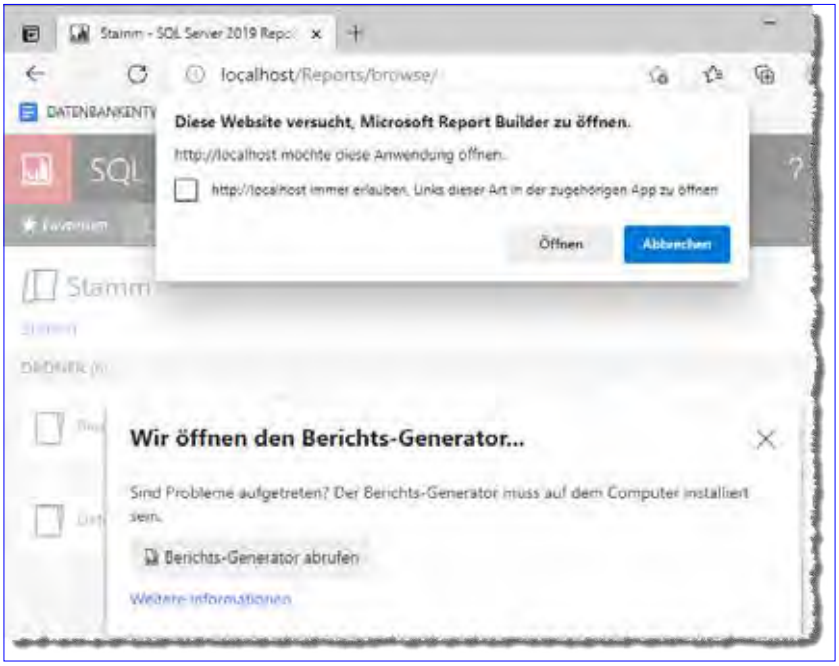


Bild 4: Freigeben des Öffnens des Report Builders

Gegebenenfalls erhalten Sie auch noch die Meldung aus Bild 5, die Sie bestätigen müssen, um die Verbindung herzustellen.

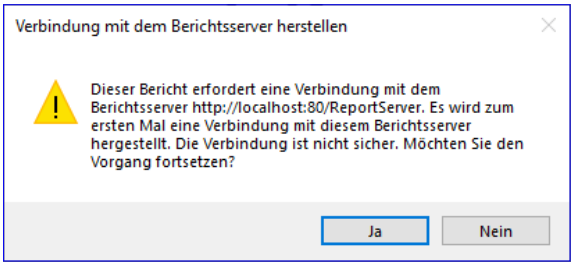


Bild 5: Sicherheitsabfrage des Berichtsservers

Danach wird der Report Builder geöffnet. Je nachdem, welchen Befehl Sie verwendet haben, zeigt der Report Builder in seinem Startbildschirm nur einen Bereich an. Im Folgenden der gewählte Befehl und der angezeigte Bereich im Startbildschirm:

- **Paginierter Bericht**: Öffnet den Report Builder mit dem Bereich **Neuer Bericht**.
- **Neues Dataset**: Öffnet den Report Builder mit dem Bereich **Neues Dataset**.

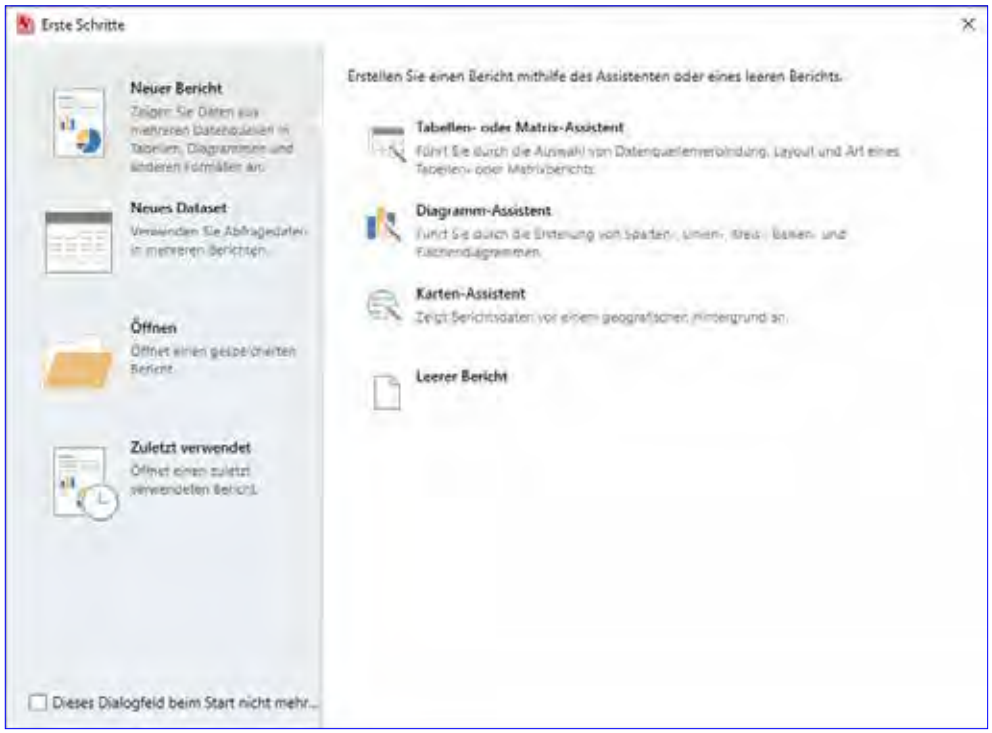


Bild 6: Der Startdialog des Report Builders

Erste Schritte im Report Builder

Der Report Builder erwartet sie mit einem aufgeräumten Startfenster. Dieses bietet nach dem Öffnen über Windows vier Bereiche an, nämlich **Neuer Bericht**, **Neues Dataset**, **Öffnen** und **Zuletzt verwendet** (siehe Bild 6) – also mehr Bereiche als beim Öffnen des Report Builders vom Web-Portal aus.

Hier finden Sie verschiedene Bereiche vor, die wiederum unterschiedliche Optionen anbieten. Da das Erstellen eines neuen Berichts für die beiden Arten **Tabellen- oder Matrix-Assistent** beziehungsweise **Diagramm-Assistent** das Vorhandensein eines Datasets erwartet, könnten wir uns hier zuerst den Bereich **Neues Dataset** ansehen. Wir können uns aber auch direkt von dem Startdialog verabschieden und die benötigen Elemente direkt über die dafür vorgesehenen Funktionen des Report Builders erstellen.

Dazu schließen Sie einfach das Startfenster und wenden sich dem Bereich Berichtsdaten zu (siehe Bild 7).

Der neue Bericht ist bereits vorhanden und wird in der Entwurfsansicht angezeigt, wir benötigen also als nächstes eine Datenquelle und ein Dataset.

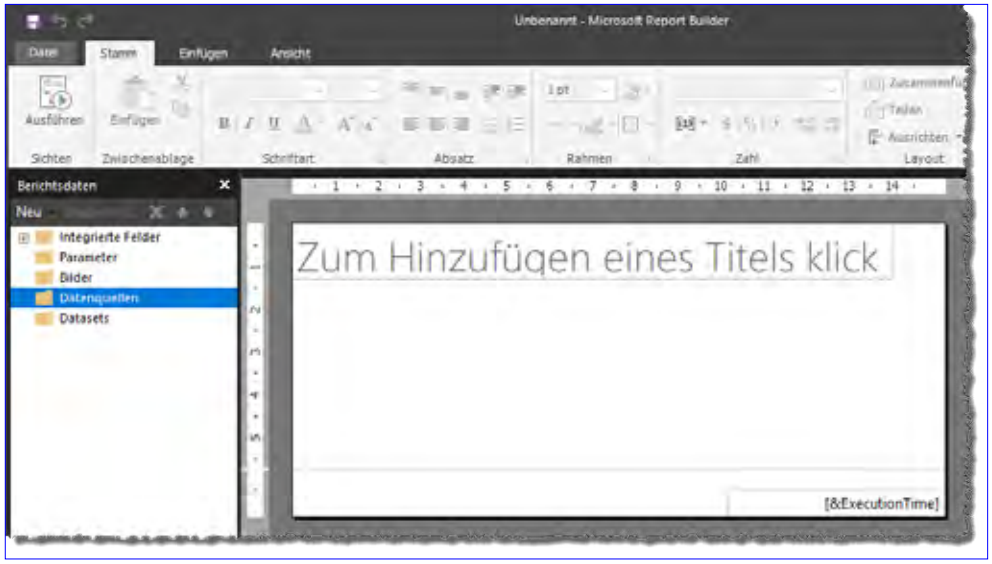


Bild 7: Startpunkt zum Erstellen von Elementen im Report Builder

Neue Datenquelle definieren

Um eine neue Datenquelle zu definieren, wählen Sie den Kontextmenü-Befehl **Datenquelle hinzufügen...** des Ordners **Datenquellen** im Bereich **Berichtsdaten** aus (siehe Bild 8).

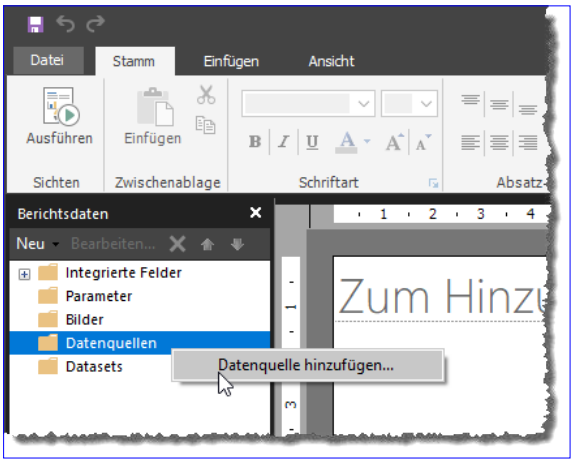


Bild 8: Hinzufügen einer Datenquelle

Im nun erscheinenden Dialog geben Sie den Namen der Datenquelle an und wählen aus, ob Sie eine freigegebene Datenquelle erstellen wollen oder eine in den Bericht eingebettete Datenquelle. Im ersten Fall haben Sie nur die Möglichkeit, über die Schaltfläche **Durchsuchen...** eine bereits freigegebene Datenquelle auszuwählen.

Wie Sie diese anlegen, zeigen wir ausführlich im Artikel **Reporting Services Datenquellen (www.datenbankentwickler.net/259)**. Sie können mit dem Report Builder also keine freigegebenen Datenquellen erstellen, sondern nur bestehende und beispielsweise mit Visual Studio erstellte Datenquellen verwenden. Also schauen wir uns nachfolgend an, wie Sie eine eingebettete Datenquelle definieren.

Dazu tragen wir zunächst den gewünschten Namen für die Datenquelle ein, in diesem Fall **Datenquelle_AdventureWorks**. Danach wechseln Sie im Dialog **Datenquelleneigenschaften** zur Option **In meinem Bericht eingebettete Verbindung verwenden** (siehe Bild 9). Wählen Sie unter **Verbindungstyp auswählen:** den Wert **Microsoft SQL Server** aus (oder einen anderen Verbindungstyp, falls gewünscht). Dann klicken Sie auf die Schaltfläche **Erstellen...** und stellen im Dialog **Connection Properties**, den wir bereits im oben genannten Artikel erläutert haben, eine Verdingung zur Datenbank **AdventureWorksLT2019** her.

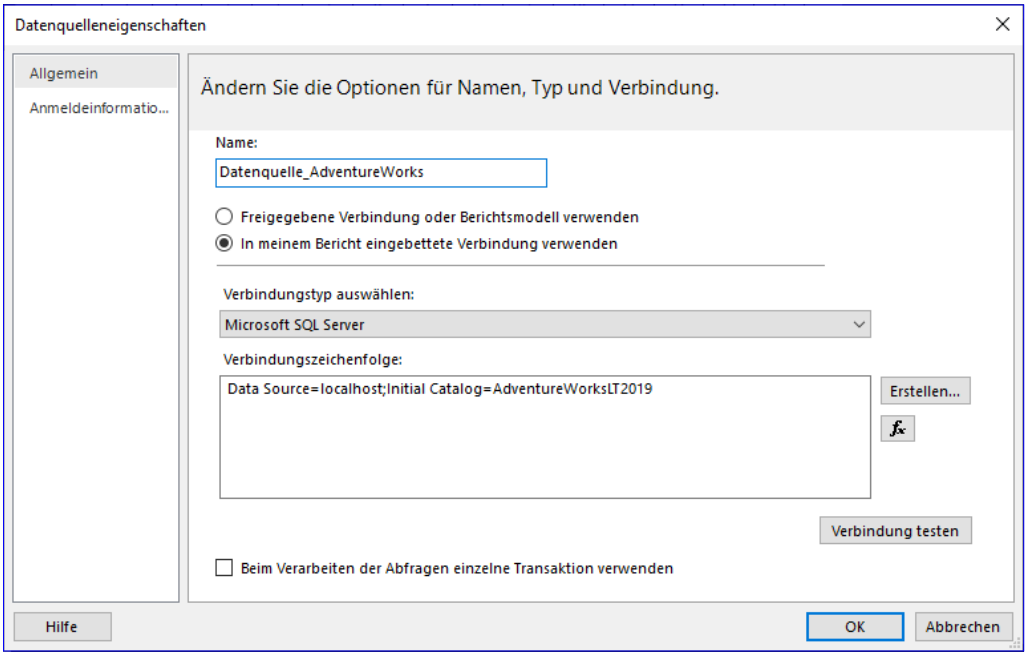


Bild 9: Neue eingebettete Datenquelle für unseren Bericht

Wie Sie diese installieren, erfahren Sie im Artikel **AdventureWorks: Schnelle Beispieldatenbank (www.datenbankentwickler.net/254)**.

In diesem Fall verwenden wir als Server **localhost**, da der Report Server auf dem lokalen Rechner läuft.

Reporting Services: Web-Portal

In den bisherigen Artikeln über die Erstellung von Berichten mit den Reporting Services haben wir einen Aspekt noch nicht betrachtet: Wie kann der Benutzer eigentlich auf die Berichte zugreifen? Immerhin benötigen wir für die Definition von Reporting Services immer eigene Projekte, die parallel beispielsweise zu unseren WPF-Anwendungen erstellt werden müssen. Dieser Artikel bringt Licht ins Dunkel und zeigt, wie Sie Berichte im Web-Portal öffnen und diesen auch noch Parameter zum Filtern von Daten übergeben können. Außerdem zeigen wir, wie Sie Berichte von WPF-Anwendungen aus aufrufen.

Veröffentlichung von Berichten vorbereiten

Die Installation der Reporting Services haben wir im Artikel **Reporting Services 2019 installieren und starten (www.datenbankentwickler.net/256)** besprochen.

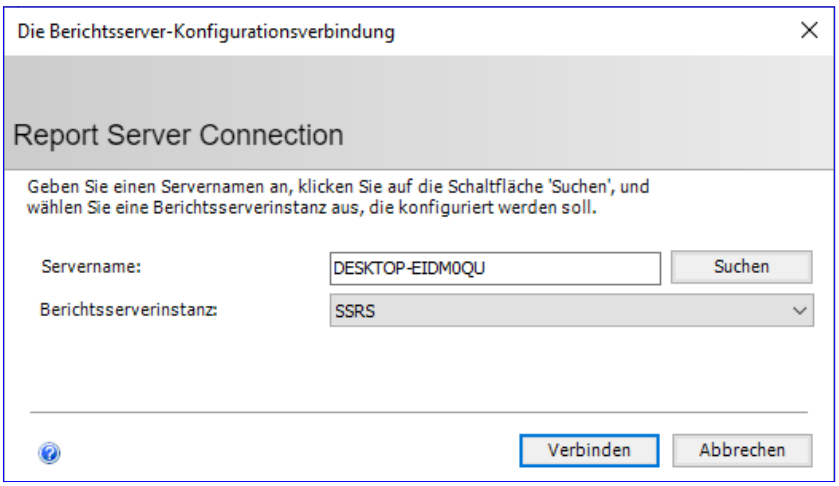


Bild 1: Erstellen einer Verbindung

Dort haben wir noch nicht die Einstellungen vorgenommen, die notwendig sind, damit Sie die von den Reporting Services generierten Berichte auch über das Web-Portal anzeigen oder von anderen Anwendungen aus abrufen können – so, wie Sie es beispielsweise von Access gewohnt sind, wo Sie einfach auf eine Schaltfläche klicken und dann der gewünschte Bericht erzeugt wird.

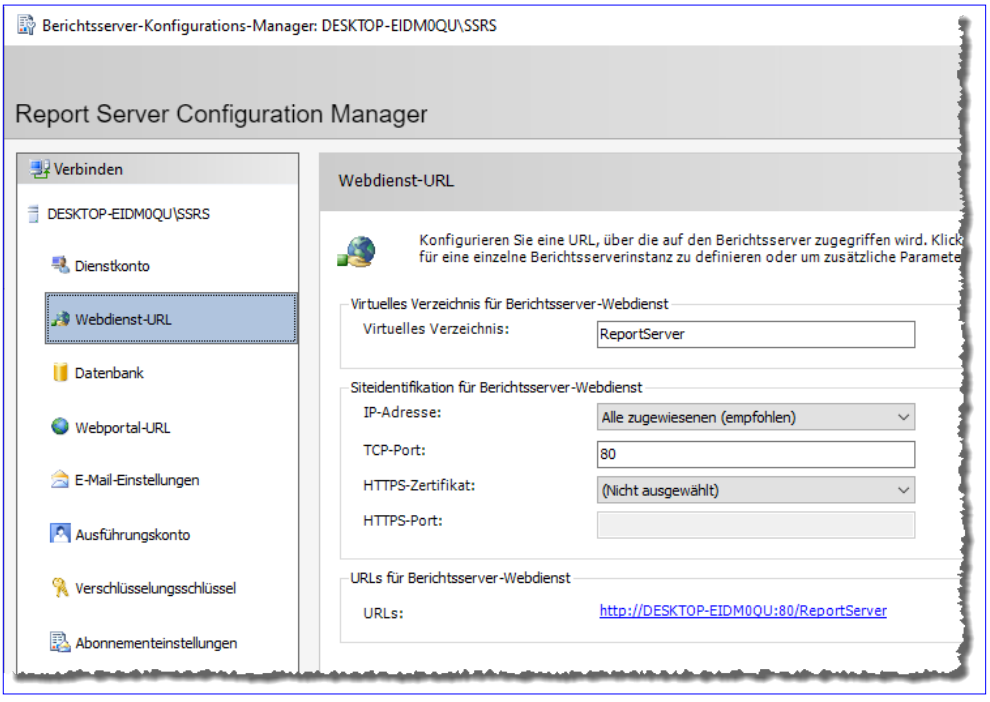


Bild 2: Einstellen der URL für den Webdienst

Um die Einstellungen nachträglich vorzunehmen, rufen Sie über die Suche von Windows die Anwendung **Report Server Configuration Manager** auf. Dieser

erwartet Sie mit dem Dialog aus Bild 1, mit dem Sie den Server und die Instanz auswählen.

Webdienst-URL erzeugen

Danach wechseln Sie im Fenster **Report Server Configuration Management** auf die Seite **Webdienst-URL** (siehe Bild 2). Hier sind bereits einige Werte voreingestellt, die Sie einfach durch Betätigen der Schaltfläche **Anwenden** übernehmen können, um den Webdienst mit diesen Werten zu initialisieren. Die wichtigste Information, die wir danach erhalten, ist die URL, die in unserem Fall wie folgt lautet:

```
http://desktop-eidm0qu/ReportServer
```

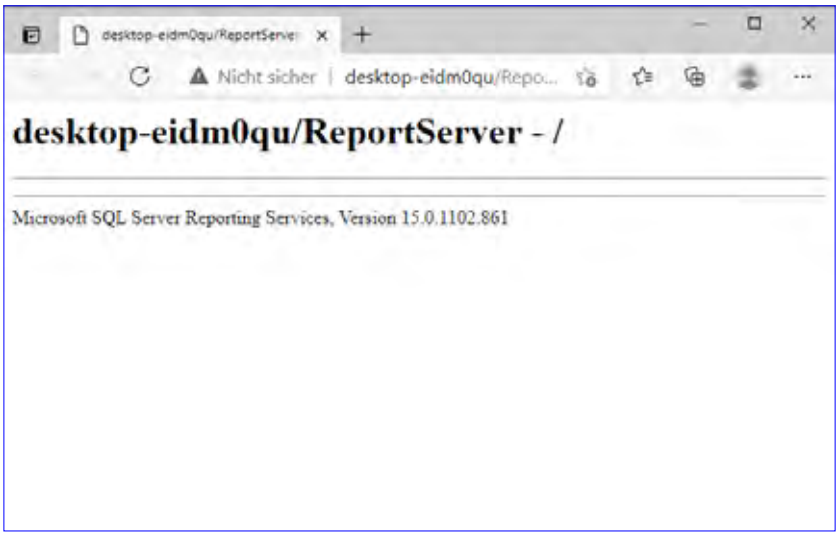


Bild 3: Erster Aufruf des Report-Servers

Die Einstellungen auf den beiden weiteren Seiten **Datenbank** und **Webportal-URL** übernehmen wir ebenfalls durch Betätigen der Schaltfläche **Anwenden**.

Erster Aufruf

Danach rufen wir erstmalig die Seite des Report-Servers auf. Dies liefert die recht leere Webseite aus Bild 3.

Berichte veröffentlichen

Um einen ersten Bericht zu veröffentlichen, öffnen Sie ein **Reporting Services Project**-Projekt – also beispielsweise das aus dem Download zum vorliegenden Artikel.

Dort klicken Sie im Projektmappen-Explorer mit

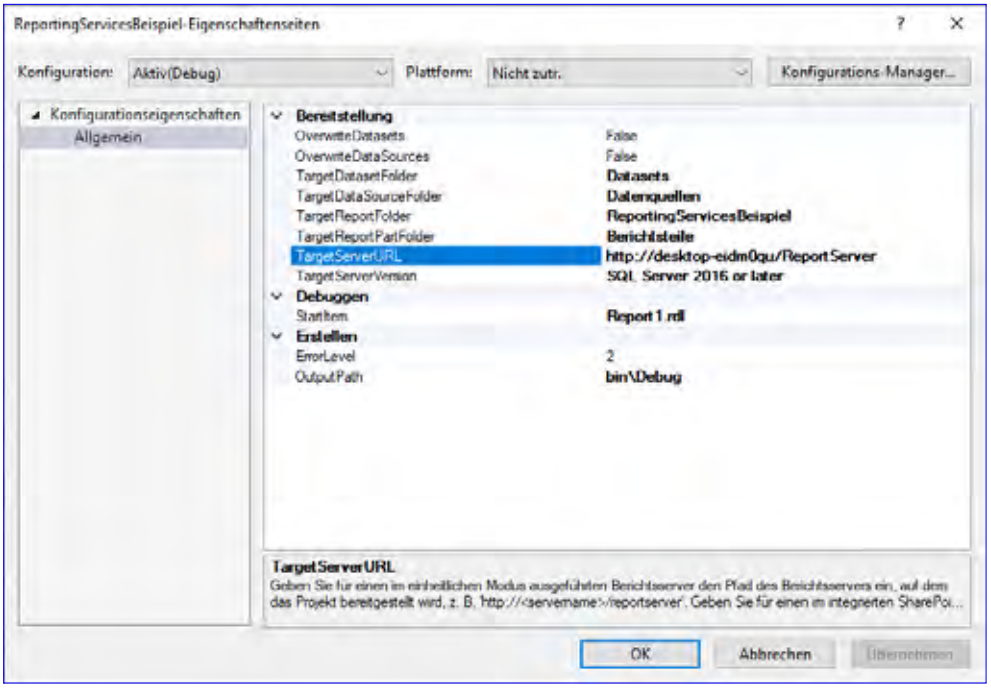


Bild 4: Einstellen der Eigenschaft **TargetServerURL**

der rechten Maustaste auf das **Projekt**-Element und wählen aus dem Kontextmenü den Eintrag **Eigenschaften** aus.

Dadurch öffnet sich der Eigenschaften-Dialog aus Bild 4. Hier finden wir unter anderem die Eigenschaft **TargetServerURL**.

Diese lautet standardmäßig **http://localhost/reportserver**.

Wenn sich der Report Server auf dem lokalen Rechner befindet, können sie diesen Wert beibehalten. Anderenfalls stellen wir den folgenden Wert ein und ersetzen dabei **<Server>** durch den Servernamen:

```
http://<Server>/Report-
Server
```

Nach dem Schließen der Eigenschaften erstellen wir das Projekt, indem wir aus dem Kontextmenü des Projekt-Elements im Projektmappen-Explorer den Eintrag **Erstellen** aufrufen.

Bild 5: Ausgabe zum Bereitstellen des Berichts

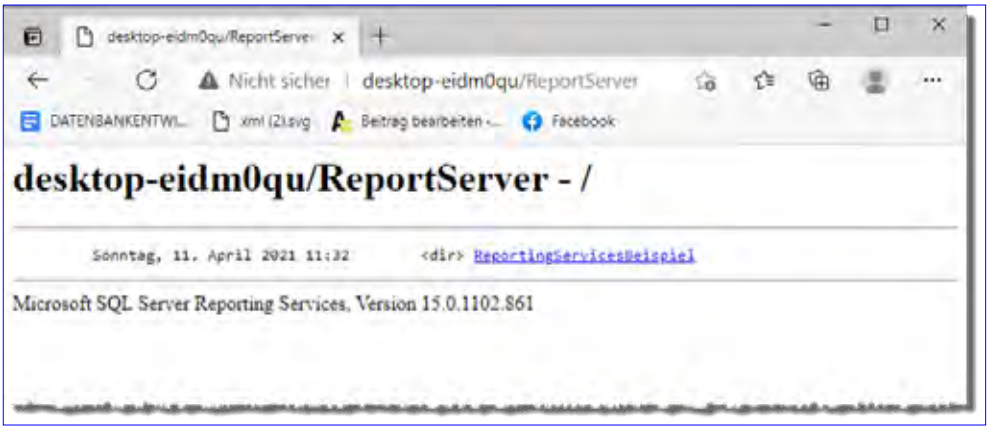


Bild 6: Anzeige des Beispielprojekts **ReportingServicesBeispiel**

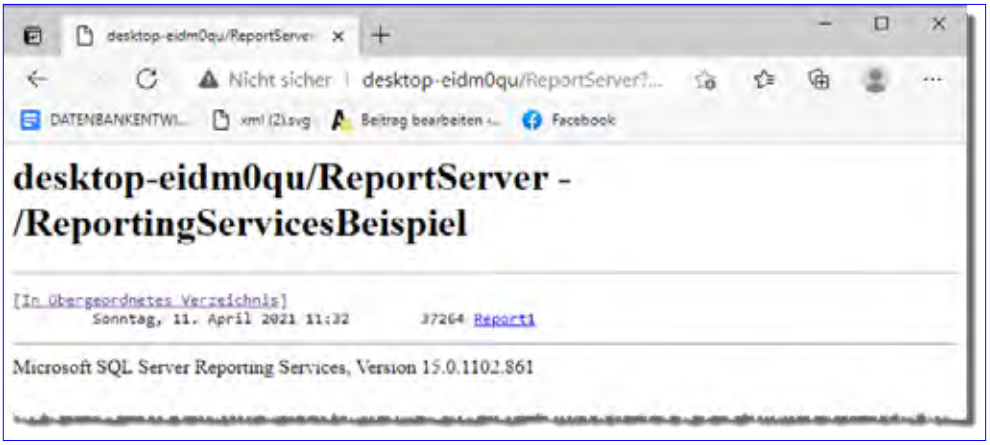


Bild 7: Anzeige des Beispielberichts **Report1**

In welcher Form wurde das Projekt nun erstellt? Wenn wir uns die Webseite der Reporting Services ansehen, finden wir dort keine neuen Elemente gegenüber dem vorherigen Zustand vor. Warum werden diese dort nicht angezeigt?

Reporting Services Datenquellen

Wenn Sie die Reporting Services nutzen wollen, um Berichte anzuzeigen, benötigen Sie dazu vor allem eines: Die Daten, die in den Berichten abgebildet werden sollen. Sie können nicht einfach wie in einer Access-Datenbank auf die Daten der aktuellen Datenbank zugreifen. Reporting Services sind vielmehr eine von den Datenquellen getrennte Instanz, der Sie erst einmal bekanntgeben müssen, welche Daten verwendet werden sollen. Das erledigen Sie mit der Definition sogenannter Datenquellen. Diese definieren Sie in verschiedenen Anwendungen. Alles rund um die Definition von Datenquellen für Reporting Services erfahren Sie in diesem Artikel!

Welche Reporting Services Datenquellen gibt es?

Wenn Sie mit Reporting Services arbeiten, sind Sie im Gegensatz zu anderen Anwendungen wie Microsoft Access, wo Tabellen und Berichte in einer einzigen Datei untergebracht sind, nicht von einer einzigen Datenquelle abhängig. Streng genommen können Sie natürlich auch in einer Access-Datenbank Daten aus verschiedenen Quellen sammeln, indem Sie entsprechende Verknüpfungen dafür hinterlegen. Mit den Reporting Services haben Sie allerdings eine eigene Instanz, die nicht speziell etwa mit einer bestimmten SQL Server-Datenbank verknüpft ist. Stattdessen können Sie die zu erstellenden Berichte und Darstellungen unabhängig von den Anwendungen, in denen Sie die Daten verwalten, zusammenführen. Dementsprechend gibt es auch separate Möglichkeiten, um diese Berichte abzurufen – in der Regel über das Web-Portal der Reporting Services.

Sie können beispielsweise die folgenden Typen von Datenquellen nutzen:

- Microsoft SQL Server- und Microsoft Azure-Datenbanken
- Oracle, SAP BW, Hyperion, SharePoint-Listen, Teradata
- OLE DB
- ODBC
- XML

Die Übersicht sieht überschaubar aus. Allerdings erhalten Sie allein mit ODBC nochmals Zugriff auf viele weitere Datenformate wie weitere Datenbanktypen, Excel-Tabellen et cetera.

Reporting Services Datenquellen mit verschiedenen Tools definieren

Microsoft bietet verschiedene Tools zum Definieren von Reporting Services Datenquellen an. Dabei handelt es sich um die folgenden – die Möglichkeiten in den verschiedenen Assistenten lassen wie einmal außen vor:

- Visual Studio: in Projekten des Typs **Berichtsserver-Assistent** sowie **Berichtsserverprojekt**

- Microsoft Report Builder
- Web-Portal der Reporting Services

In den folgenden Abschnitten schauen wir uns die verschiedenen Möglichkeiten im Detail an.

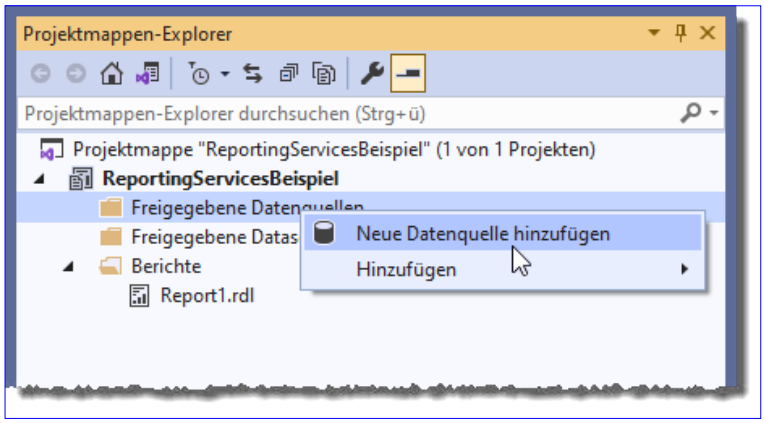


Bild 1: Hinzufügen einer Datenquelle in Visual Studio

Eingebettete oder freigegebene Datenquellen

Es gibt zwei Arten von Datenquellen:

- Bei eingebetteten Datenquellen werden die Informationen mit der Definition des Berichts gespeichert. Sie kann dementsprechend auch nur von diesem Bericht verwendet werden.
- Bei freigegebenene Datenquellen sind die Definitionen der Datenquelle für alle Berichte verfügbar, die sich im gleichen Projekt befinden. Sie können freigegebene Datenquellen allerdings nicht von anderen Projekten aus verwenden.

Weiter unten zeigen wir, wie Sie die beiden Arten definieren können.

Reporting Services Datenquellen mit Visual Studio definieren

In Visual Studio definieren Sie Datenquellen für die Reporting Services in Projekten des Typs **Berichtsserverprojekt-Assistent** oder **Berichtsserver-Projekt**.

Nach dem Durchlaufen des Assistenten oder nach dem Öffnen eines Berichtsserver-Projekts ohne Assistent finden Sie im Projektmappen-Explorer einen Ordner namens **Freigegebene**

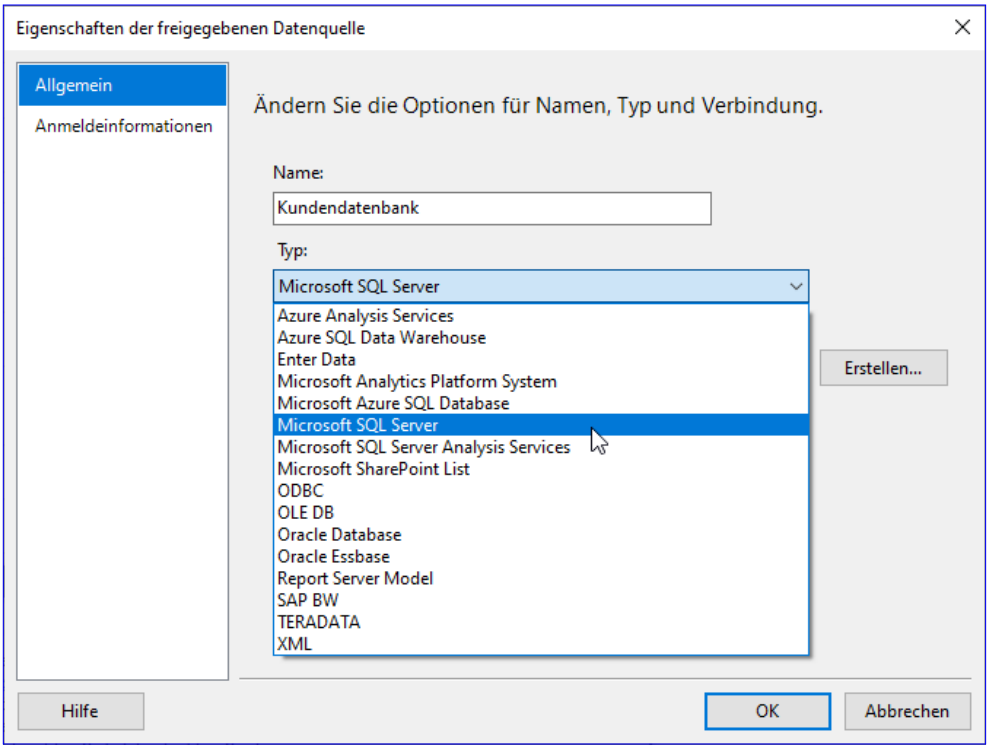


Bild 2: Auswahl des Typs der Datenquelle

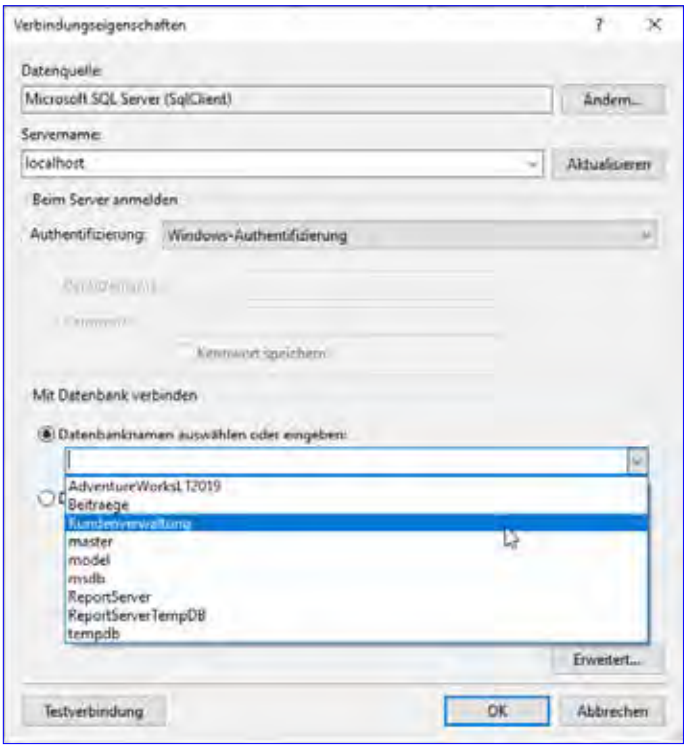


Bild 3: Definieren der Verbindungseigenschaften

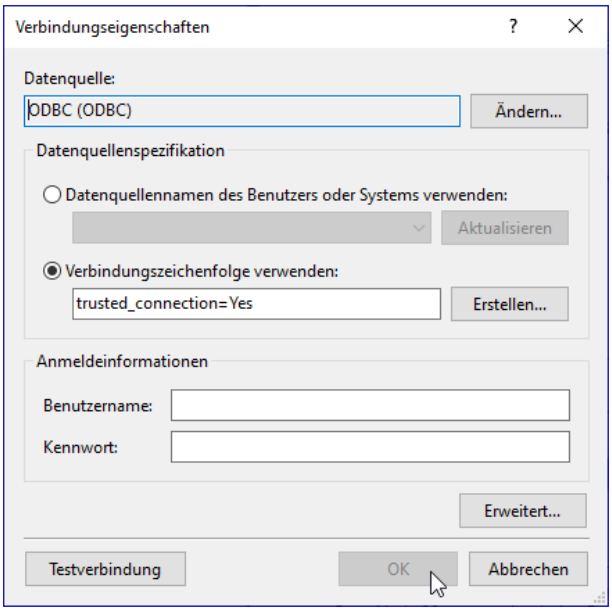


Bild 4: Verbindungseigenschaften für eine ODBC-Datenquelle

Datenquellen vor. Mit einem Rechtsklick auf diesen Ordner zeigen Sie den Kontextmenübefehl **Neue Datenquelle hinzufügen** an (siehe Bild 1).

Freigegebene Datenquellen in Visual Studio erstellen

Anschließend erscheint der Dialog **Eigenschaften der freigegebenen Datenquelle** (siehe Bild 2). Hier geben Sie den Namen der Datenquelle ein und wählen den Typ aus – in diesem Fall **Microsoft SQL Server**. Anschließend klicken Sie auf die Schaltfläche **Erstellen...**, um die Details festzulegen.

Dies liefert den Dialog **Verbindungseigenschaften**, den Sie bereits von anderen Anwendungen kennen dürften. Dieser wird je nach Auswahl des Typs der Datenquelle mit anderen Eigenschaften angezeigt. Für eine SQL Server-Datenbank sieht der Dialog beispielsweise wie in Bild 3 aus.

Wenn Sie hingegen eine ODBC-Datenquelle nutzen wollen und dazu im Dialog **Eigenschaften der freigegebenen Datenquelle** den Eintrag **ODBC** im Feld **Typ** auswählen,

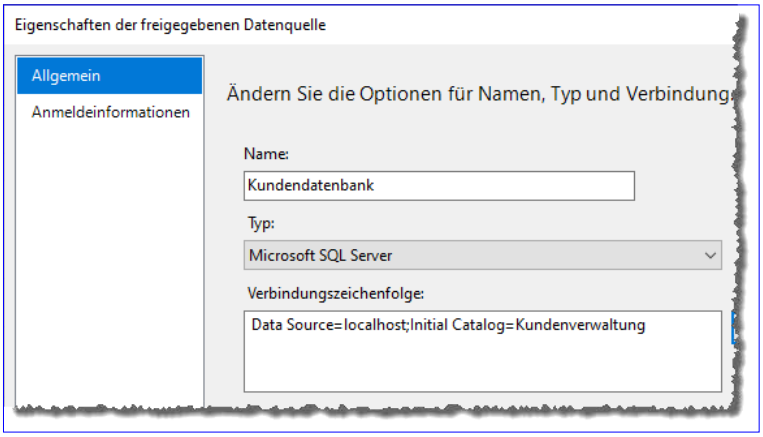


Bild 5: Verbindungszeichenfolge für eine SQL Server-Datenbank

erhalten Sie den Dialog aus Bild 4 für die Eingabe der Verbindungseigenschaften einer ODBC-Datenquelle.

Für manche Datenquellentypen gibt es auch gar keinen Dialog, sodass Sie die Verbindungszeichenfolge manuell eintragen müssen – beispielsweise beim Typ **XML**.

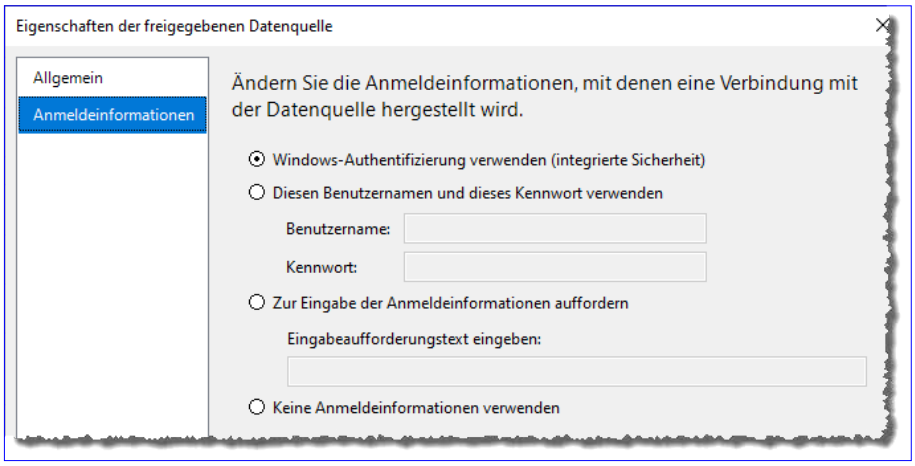


Bild 6: Einstellungen für die Anmeldeinformationen

SQL Server-Datenquelle

Wenn Sie mit dem Dialog **Verbindungseigenschaften** die Daten für eine SQL Server-Datenbank erfassen und das schnell erledigen wollen, haben Sie am besten den Namen des SQL Servers bereits parat. Das Ermitteln der verfügbaren Server mit der Schaltfläche Aktualisieren dauert immer eine Weile.

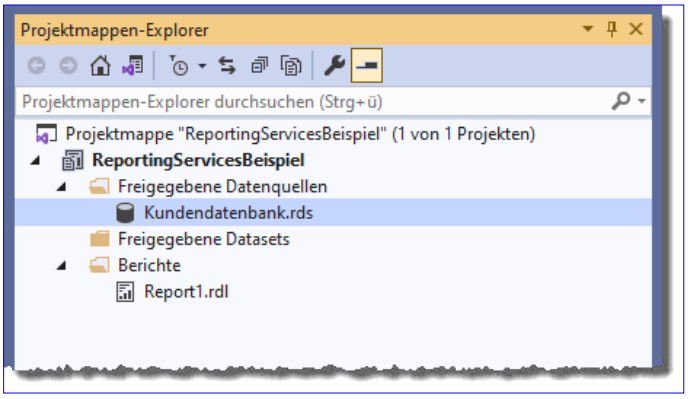


Bild 7: Neue Datenquelle für die Reporting Services

Wenn Sie jedoch den Namen direkt eintragen, indem Sie diesen entweder auswendig kennen (manchmal reicht ja auch **localhost**) oder aus dem SQL Server Management Studio kopieren, finden Sie schnell alle verfügbaren Datenbanken in der Liste unter **Mit Datenbank verbinden|Datenbanknamen auswählen oder eingeben** vor.

Wählen wir hier beispielsweise den Namen **Kundenverwaltung** aus und schließen den Dialog nach erfolgreichem Test mit der Schaltfläche **Testverbindung**, erhalten Sie die Verbindungszeichenfolge aus Bild 5.

Anmeldeinformationen festlegen

Anschließend stellen Sie im gleichnamigen Bereich noch die **Anmeldeinformationen** ein (siehe Bild 6). Hier behalten wir die Option **Windows-Authentifizierung verwenden (integrierte Sicherheit)** bei. Damit können wir die aktuellen Windows-Benutzer auf die verschiedenen Benutzergruppen/Benutzer mit den entsprechenden Berechtigungen mappen. Sie können auch Benutzername/Kennwort verwenden oder zur Eingabe von Anmeldeinformationen auffordern.

Nach dem Anlegen der Reporting Services Datenquelle

Haben Sie das Anlegen abgeschlossen, finden Sie die neue Datenquelle als Eintrag im Ordner **Freigegebene Datenquellen** vor (siehe Bild 7). Reporting Services Datenquellen haben eine Bezeichnung mit der Endung **.rds**.

Reporting Services Datasets

Wenn Sie ein Reporting Services Projekt erstellt haben, benötigen Sie vor dem Anlegen von Berichten zwei Dinge: eine Datenquelle, auch Datasource genannt, und eine Art Abfrage, wie bei den Reporting Services Datasets genannt werden. Wie Sie Datenquellen auf verschiedene Arten erstellen, haben wir im Artikel »Reporting Services Datenquellen« erläutert. Im vorliegenden Artikel beschreiben wir, wie Sie mithilfe von Datasets die Daten für Ihren Bericht zusammenstellen und welche unterschiedlichen Möglichkeiten es dafür gibt. Dabei schauen wir uns wieder Visual Studio, den SQL Server Report Builder und das Web-Portal der Reporting Services an.

Freigegebene Datasets oder eingebettete Datasets?

Genau wie bei den Datenquellen beziehungsweise Datasources gibt es auch bei den Datasets zwei Varianten – nämlich freigegebene Datasets und eingebettete Datasets. Die freigegebenen Datasets können Sie in einem Projekt oder im Web-Portal erstellen. Eine wichtige Voraussetzung für ein freigegebenes Dataset ist das Vorhandensein einer freigegebenen Datenquelle. Für ein eingebettetes Dataset können Sie eine freigegebene Datenquelle verwenden, aber auch eine eingebettete Datenquelle.

Welchen Typ von Dataset sollten Sie nun verwenden? Es verhält sich in einem Aspekt ähnlich wie bei Microsoft Access: Dort konnten Sie Abfragen als eigene Objekte speichern und diese dann Formularen über die Eigenschaft **Datensatzquelle** oder Listensteuerelementen über die Eigenschaft **Datensatzherkunft** zuweisen. Oder Sie haben die als Datenquelle zu verwendende Abfrage als SQL-Abfrage direkt für die jeweilige Eigenschaft hinterlegt. Beides hat Vor- und Nachteile: Wenn Sie die Abfrage als eigenes Objekt speichern, können Sie diese als Datenquelle für mehr als ein Formular oder einen Bericht und gleichzeitig noch als Datenquelle für Listensteuerelemente nutzen.

Der Vorteil und der Nachteil ist gleichzeitig, dass sich Änderungen an der Abfrage auf alle referenzierenden Elemente auswirkt. Andersherum sammeln Sie eine Menge Datenquellen in Formularen, Berichten und Steuerelementen an, die nur schwer zu warten sind.

Eine Alternative ist es, alle Abfragen als eigene Objekte zu speichern und diese den Objekten beziehungsweise Steuerelementen zuzuweisen. Freigegebene Datasets haben den Vorteil, dass diese gecached werden können.

Und es gibt noch einen gravierenden Unterschied von freigegebenen Dataset gegenüber den erwähnten Abfragen von Access: Sie können diese nämlich auch von anderen Projekten aus verwenden!

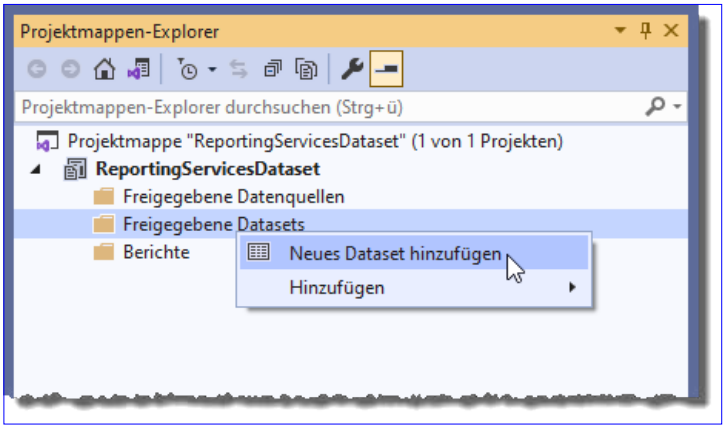


Bild 1: Hinzufügen eines neuen Datasets per Kontextmenü

Freigegebenes Reporting Services Dataset mit Visual Studio

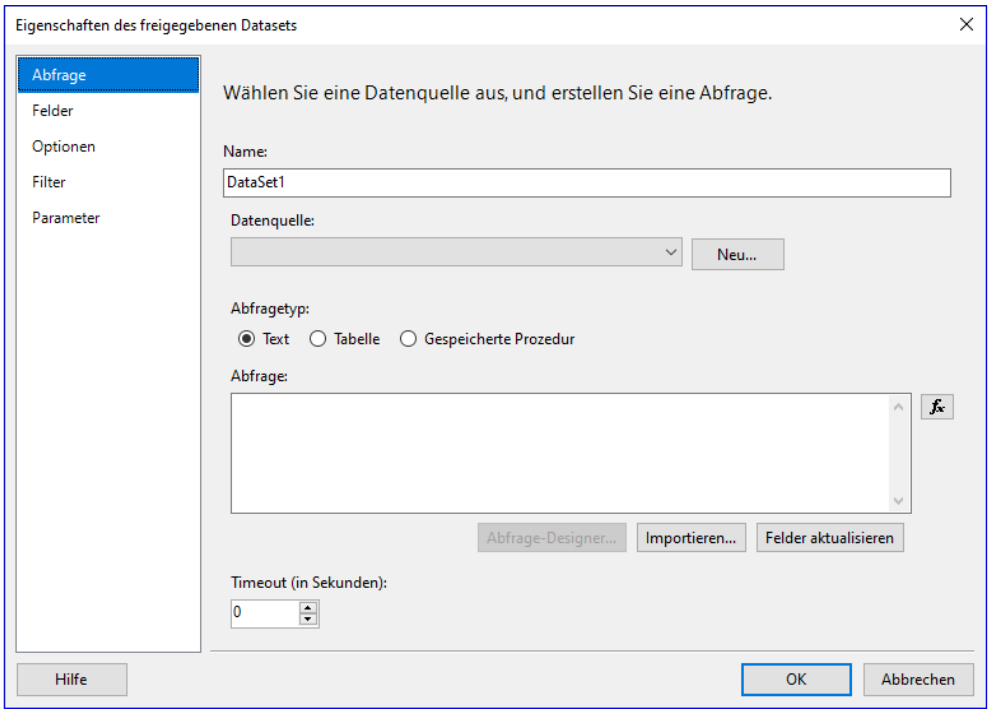


Bild 2: Dialog zum Anlegen eines neuen Datasets

Wenn Sie in Visual Studio ein neues Projekt des Typs **Berichtsserverprojekt-Assistent** oder **Berichtsserverprojekt** erstellt haben, finden Sie im Projektmappen-Explorer neben **Freigegebene Datenquellen** einen weiteren Ordner namens **Freigegebene Datasets** vor. Wollen Sie für diesen ein neues freigegebenes Dataset hinzufügen, betätigen Sie den Kontextmenü-Befehl **Neues Dataset hinzufügen** dieses Eintrags (siehe Bild 1). Hierbei ist es wichtig zu wissen, dass Sie auf jeden Fall zuvor mindestens eine freigegebene Datenquelle benötigen. Diese können Sie, wie im Artikel **Reporting Services Datenquellen (www.datenbankentwickler.net/259)** beschrieben erstellen – oder Sie klicken einfach auf **Neues Dataset hinzufügen** und legen die Datenquelle über die dortige Funktion an.

Der Kontextmenü-Eintrag öffnet den Dialog **Eigenschaften des freigegebenen Datasets**. Hier legen Sie den Namen des Datasets fest. Außerdem wählen Sie die Datenquelle aus den freigegebenen Datenquellen aus. Sollte noch keine freigegebene Datenquelle vorhanden sein oder wollen Sie eine neue Datenquelle erstellen, können Sie das mit einem Klick auf die Schaltfläche **Neu...** erledigen. Das liefert den Dialog **Eigenschaften der freigegebenen Datenquelle**, den wir im oben genannten Artikel im Detail beschrieben haben (siehe Bild 2).

An dieser Stelle gehen wir davon aus, dass Sie gemäß der Anleitung in dem oben erwähnten Artikel bereits eine Datenquelle erstellt haben.

Nun schauen wir uns den Abfragetyp an. Hier gibt es drei Möglichkeiten:

- **Text**: Eingabe einer Abfrage im SQL-Format. Dazu können Sie verschiedene Tools nutzen, die wir im Anschluss vorstellen werden.

- **Tabelle**: Erlaubt die Auswahl einer Tabelle. Diese Option steht jedoch nur in ODBC- oder OLE DB-Datenquellen zur Verfügung.

- **Gespeicherte Prozedur**: Wenn Sie die Daten für den Bericht bereits in einer gespeicherten Prozedur vorbereitet haben, können Sie diese aus der Liste der gespeicherten Prozeduren der in der Datenquelle angegebenen Datenbank auswählen.

SQL-Abfrage als Quelle für ein Dataset definieren

Wenn Sie unter Abfragetyp die Option **Text** wählen, können Sie verschiedene Optionen zum Erstellen der Abfrage nutzen:

- **Abfrage-Designer...**: Dies öffnet den Abfrage-Designer, mit dem Sie ähnlich wie im Abfrageentwurf von Access die Datenquelle zusammenklicken können.

- **Importieren...**: Öffnet einen Dateiauswahl-Dialog, mit dem Sie eine zuvor gespeicherte Abfragedatei mit der Dateiendung **.sql** oder **.rdl** auswählen können.

- **Felder aktualisieren**: Aktualisiert die Liste der Felder im Bereich **Felder** des Dialogs.

- **fx**: Diese Schaltfläche öffnet einen Dialog, mit dem Sie den aktuellen Ausdruck bearbeiten und um weitere Informationen anreichern können.

SQL-Abfrage per Abfrage-Designer erstellen

Wählen Sie die Option **Abfrage-Designer...**, erscheint der Dialog aus Bild 3. Mit diesem fügen Sie zunächst die Tabellen hinzu, deren Daten im Bericht angezeigt werden sollen. Dann markieren Sie die Felder, die im Abfrageergebnis erscheinen sollen. Mit einem Klick auf die Ausrufezeichen-Schaltfläche können Sie das Abfrageergebnis vor-

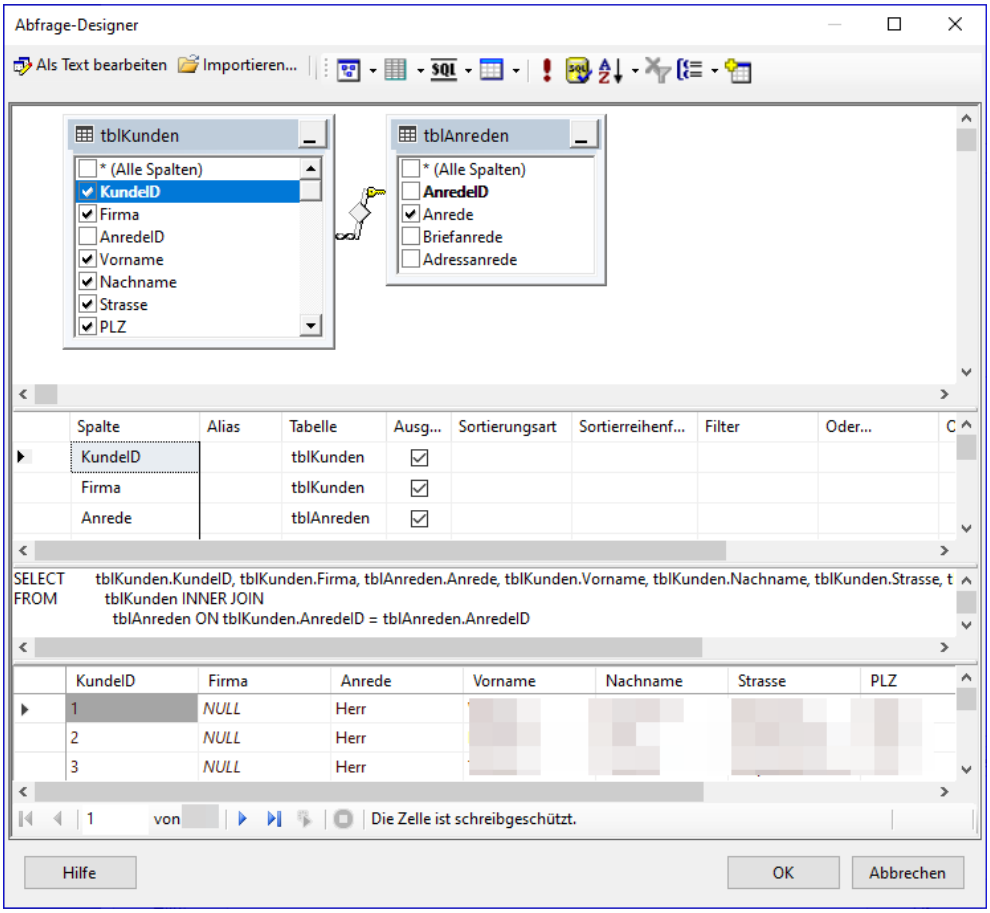


Bild 3: Entwurfsansicht eines Datasets

ab anzeigen lassen. In der Liste der anzuzeigenden Felder können Sie noch Sortierungen und Filterkriterien festlegen.

SQL-Abfrage bearbeiten

Die zuvor zusammengestellte SQL-Abfrage können Sie durch einen Klick auf die Schaltfläche mit der Beschriftung **fx** im Dialog **Ausdruck** anzeigen (siehe Bild 4). Dieser Dialog zeigt den SQL-Ausdruck in einem Textfeld an, wo Sie diesen manuell bearbeiten können.

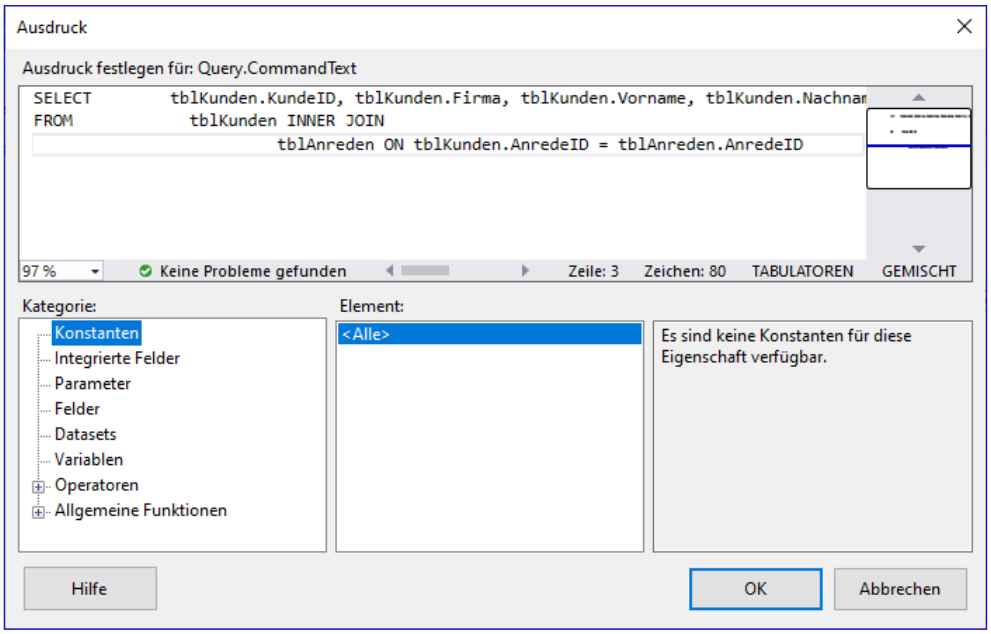


Bild 4: Bearbeiten des Abfrage-Ausdrucks

Darunter finden Sie eine Liste mit Kategorien, nach deren Auswahl die Elemente der jeweiligen Kategorie rechts in der Liste mit der Beschriftung **Element** angezeigt werden. Diese können Sie allerdings an dieser Stelle nicht nutzen, da ein Hinzufügen nur per Doppelklick auf das jeweilige Element möglich ist, was aber dann den aktuellen Inhalt des Ausdrucks durch den angeklickten Eintrag ersetzt.

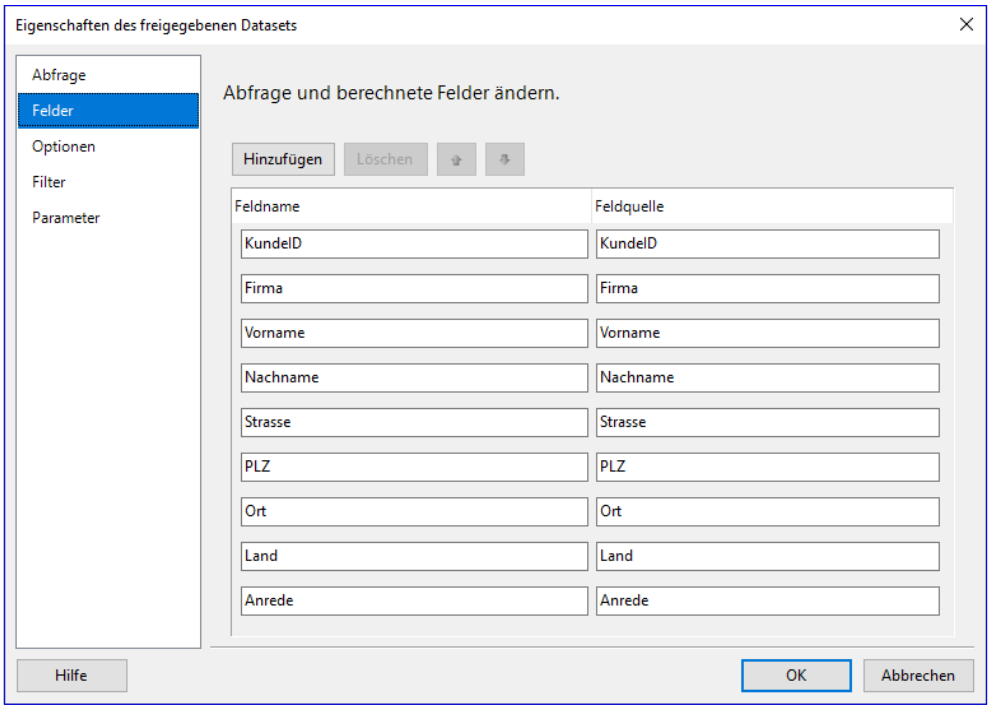


Bild 5: Bearbeiten der Felder der Abfrage

Der Bereich Felder in den Dataset-Eigenschaften

Wechseln Sie im Fenster **Eigenschaften des freigegebenen Datasets** zum Bereich **Felder**, zeigt der Dialog die Liste der Feldnamen und eine dazu passende Feldquelle an (siehe Bild 5). Hier finden Sie die über den Abfrage-

entwurf hinzugefügten Felder. Sie können aber noch weitere Felder hinzufügen und damit weitere Informationen für die Verarbeitung in den Berichen bereitstellen. Dazu stellt der Dialog verschiedene Schaltflächen zur Verfügung, zum Beispiel **Hinzufügen** und **Löschen** und zwei Schaltflächen zum Ändern der Reihenfolge der angezeigten Felder.

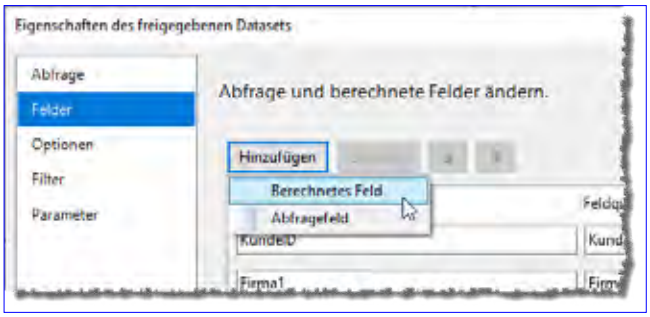


Bild 6: Hinzufügen von Feldern zum Dataset

Klicken Sie auf die Schaltfläche **Hinzufügen**, erscheinen zwei weitere Befehle (siehe Bild 6):

- **Berechnetes Feld**: Fügt eine neue Zeile hinzu, mit der Sie über die Schaltfläche fx den gewünschten berechneten Ausdruck zusammenstellen können.
- **Abfragefeld**: Fügt ein neues Feld hinzu, das Sie manuell ausfüllen müssen.

Bild 7: Einfügen eines berechneten Feldes

Berechnete Felder zu Dataset hinzufügen

Ein neues, berechnetes Feld sieht wie in Bild 7 aus.

Klicken Sie auf die Schaltfläche **fx**, erscheint der Dialog **Ausdruck**. An dieser Stelle, also beim Anlegen oder Bearbeiten eines Datasets, sind die Möglichkeiten dieses Dialogs begrenzt, denn das Dataset ist noch keinem Bericht zugeordnet. Daher können wir, auch wenn wir bereits eine SQL-Abfrage als Datenquelle festgelegt haben, beispielsweise noch nicht auf die Felder dieser Abfrage zugreifen, um diese in Ausdrücken zu verwenden.

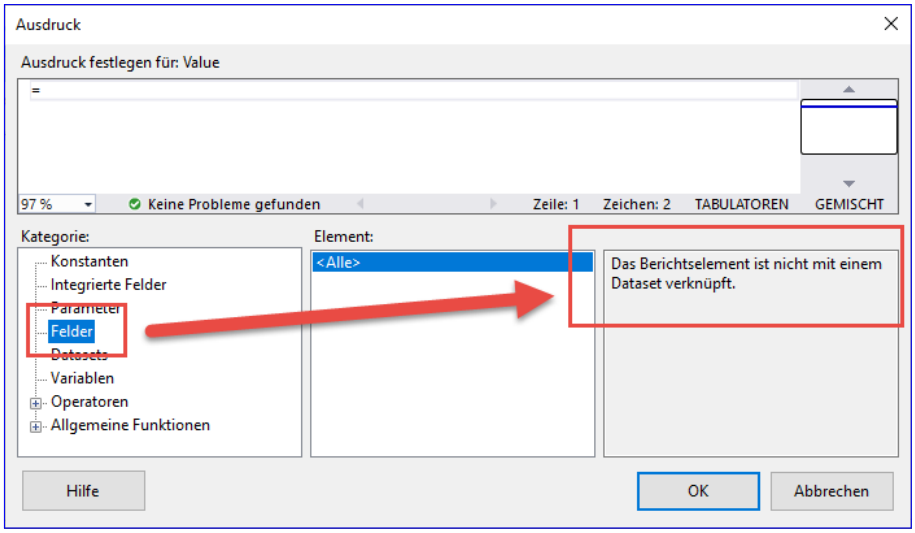


Bild 8: Für freigegebene Datasets fehlen wichtige Elemente.

Daher erscheint beispielsweise für die Kategorie Felder der Text **Das Berichtselement ist**

nicht mit einem Dataset verknüpft (siehe Bild 8).

Also machen wir einen Zwischenschritt und erstellen einen neuen Bericht. Dazu wählen wir den Kontextmenü-Eintrag **Hinzufügen|Neues Element...** des Projekt-Elements im Projektmappen-Explorer aus und legen im Dialog **Neues Element hinzufügen** einen neuen Bericht namens **KundenMitAnrede.rdl** an. Für diesen Bericht fügen wir dann im Bereich **Berichtsdaten** (einzublenden mit dem Menübefehl **Ansicht|Berichtsdaten**) eine neue Datenquelle hinzu. Das erledigen wir mit dem Kontextmenü-Befehl **Datenquelle hinzufügen...** des Ordners **Datenquellen**. Die neue Datenquelle soll **Kundenverwaltung_KundenMitAnrede** heißen und die freigegebene Datenquelle **Kundenverwaltung** nutzen (siehe Bild 9).

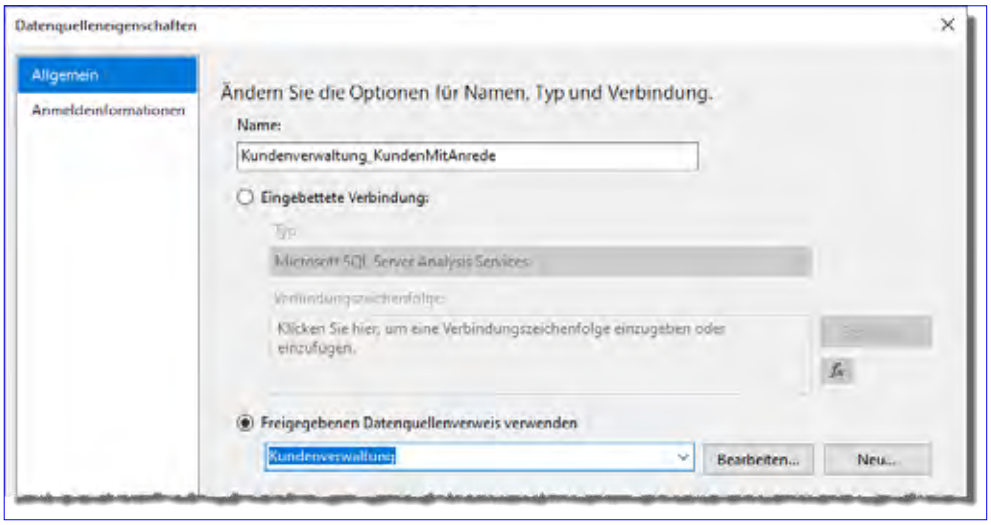


Bild 9: Hinzufügen der eingebetteten Datenquelle zu einem Bericht

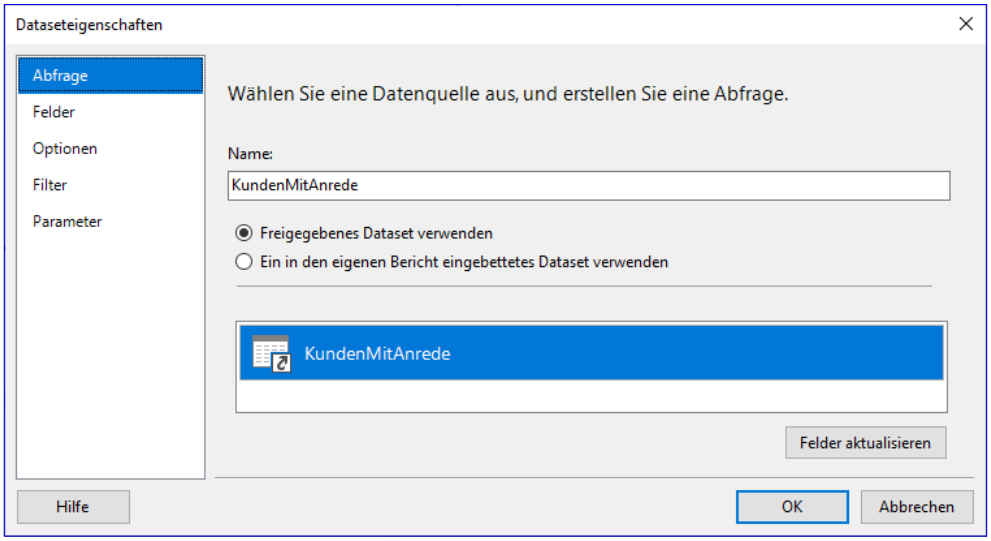


Bild 10: Auswählen des freigegebenen Datasets als Dataset des Berichts

Nach dem Schließen des Dialogs öffnen wir über den Kontextmenü-Befehl **Dataset hinzufügen...** des Ordners **Datasets** den Dialog **Dataseteigenschaften**. Dieser zeigt direkt das einzige freigegebene Dataset der verwendeten freigegebenen Datenquelle an, die wir nun auswählen (siehe Bild 10).

Wo der Dialog bereits geöffnet ist, wechseln wir direkt zum Bereich **Felder**, der uns gerade noch nicht alle Funktionen offenbart hat. Klicken wir dort auf **Hinzufügen|Berechnetes Feld** und dann für das neue Feld auf die Schaltfläche **fx**, finden wir unter der Kategorie **Felder** immer noch nicht die Felder der Abfrage vor. Dazu müssen wir den Dialog erst schließen und so das Dataset dem aktuellen Bericht zuweisen. Erst nach einem erneuten Öffnen des Datasets, dem Anklicken von **Hinzufügen|Berechnetes Feld** im Bereich **Felder** und einem Klick auf die Schaltfläche **fx** für das neue

Element, finden wir unter der Kategorie **Felder** die Felder der Abfrage vor (siehe Bild 11). Damit können wir nun Ausdrücke auf Basis der Felder zusammenstellen. So können wir beispielsweise PLZ und Ort zu einem einzigen Feld zusammenführen:

```
=Fields!PLZ.Value & " "
& Fields!Ort.Value
```

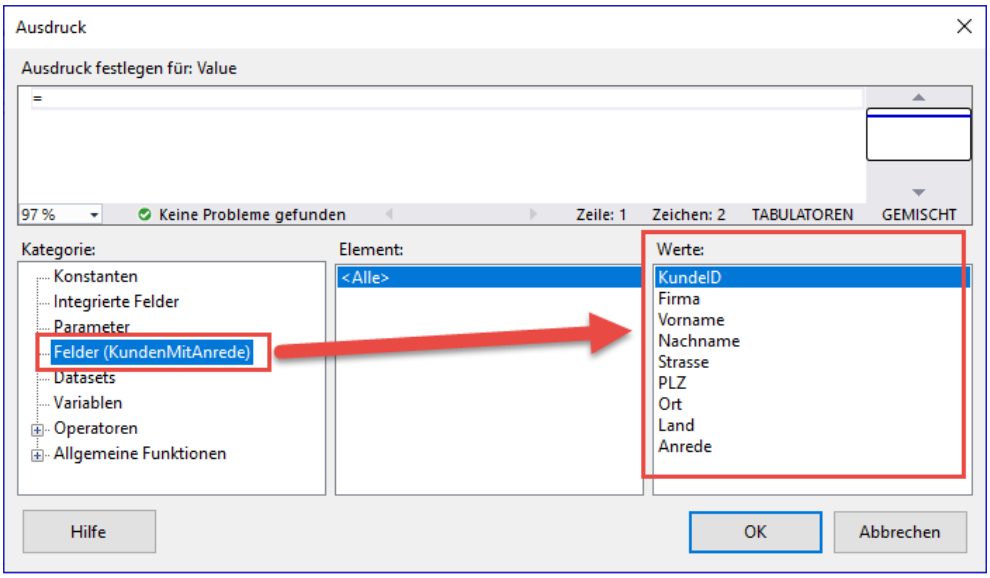


Bild 11: Zugriff auf die Felder eines Datasets für das Definieren eines Ausdrucks

Im Dialog **Ausdruck positionieren** Sie dazu die Einfügemarke jeweils an der entsprechenden Stelle unter **Ausdruck festlegen für: Value** und klicken dann doppelt auf die Elemente der Liste **Werte:**, um diese zum Ausdruck hinzuzufügen (siehe Bild 12). Im Ausdruck können Sie auch Visual Basic-Befehle verwenden. So können Sie beispielsweise Teile von Datumsfeldern mit Funktionen wie **Day**, **Month** oder **Year** extrahieren.

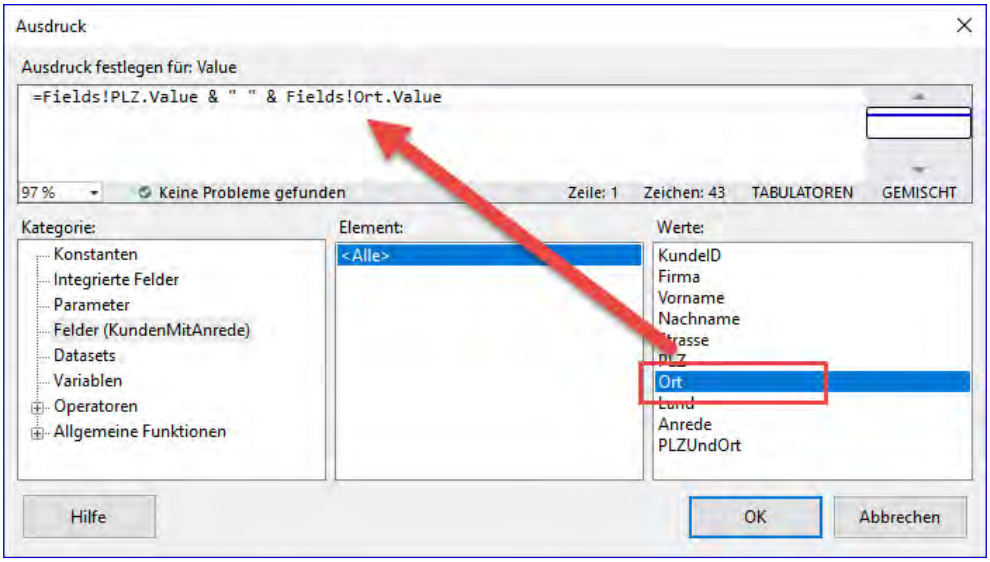


Bild 12: Zusammenstellen eines berechneten Ausdrucks

Benutzerdefinierte Visual Basic-Funktionen in Ausdrücken

Sie können sogar eigene Visual Basic-Funktionen definieren und diese zur Berechnung von Ausdrücken heranziehen. Diese Möglichkeit werden wir in einem Artikel namens **Reporting Services Berichte mit VB-Funktionen** (**www.datenbankentwickler.net/264**) vorstellen.

Filter verwenden

Im Bereich **Filter** des Dialogs **Dataseteigenschaften** legen Sie Filterkriterien für das Dataset fest. Hier finden Sie zunächst nur die aktivierte Schaltfläche **Hinzufügen** vor. Diese zeigt einen neuen, leeren Block für ein Filter-

Reporting Services Datasets mit Parametern

Ein Dataset, das als Datenquelle eines Berichts der Reporting Services dient, ist selten statisch aufgebaut – also so, dass es immer alle Daten einer Tabelle oder Abfrage liefert oder einen bestimmten Teil. Stattdessen sollen Berichte oft Daten liefern, die sich auf einen bestimmten Zeitraum beziehen, auf einen bestimmten Kunden oder auch auf bestimmte Artikel oder Kategorien. In den bisherigen Artikeln haben wir erläutert, wie Sie Berichte auf Basis statischer Datenquellen erstellen. Der vorliegende Artikel fügt die Verwendung von Parametern mit hinzu, damit der Benutzer genau festlegen kann, welche Daten der von ihm gewählte Bericht liefern soll.

Vorbereitung für diesen Artikel

Da wir uns explizit um den Einsatz von Parametern kümmern wollen, gehen wir von einem Visual Studio-Projekt des Typs **Berichtsserverprojekt** namens **DatasetMitParametern** aus.

Diesem haben wir eine freigegebene Datenquelle namens **AdventureWorks** auf Basis der SQL Server-Datenbank **AdventureWorksLT2019** hinzugefügt. Wo Sie diese Beispieldatenbank finden und wie Sie diese installieren, erfahren Sie im Artikel **AdventureWorks: Schnelle Beispieldatenbank (www.datenbankentwickler.net/254)**.

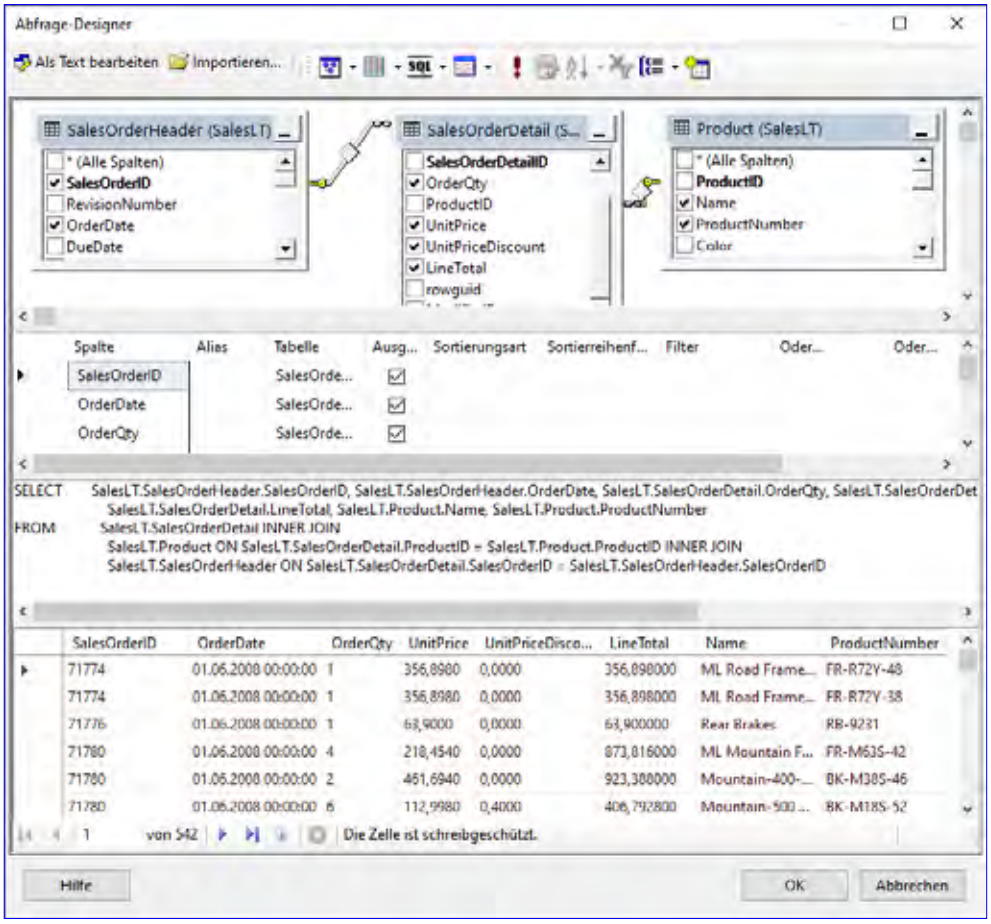


Bild 1: Abfrage für das Dataset **ProductsSalesOrders**

Dataset anlegen

Das Dataset für diesen Bericht legen wir als eingebettetes Dataset an. Damit erhalten wir zum Beispiel den Vorteil, dass die in diesem Dataset definierten Parameter direkt vom Bericht als solche erkannt werden.

Bericht anlegen

Nun legen wir einen neuen Bericht namens **SalesPerProduct.rdl** an. Diesem weisen wir die zuvor angelegte freigegebene Datenquelle **AdventureWorks.rds** als Datenquelle zu und geben auch der neuen eingebetteten Datenquelle den Namen **AdventureWorks**.

Dann legen wir für den Bericht ein neues eingebettetes Dataset namens **ProductsSalesOrders** an (siehe Bild 1). Das Dataset verwendet einige Felder der drei Tabellen **Product**, **SalesOrderDetail** und **SalesOrderHeader** als Datenquelle. Damit haben wir ausreichend Daten zum Demonstrieren der Parameter-Funktionen der Reporting Services.

Der Bereich **Berichtsdaten** sieht anschließend wie in Bild 2 aus.

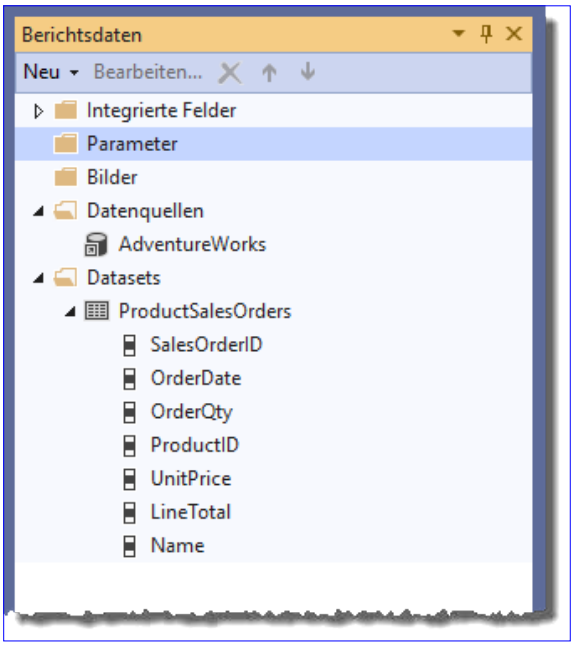


Bild 2: Daten für den Bericht

Nun blenden wir die Toolbox ein und fügen dem Bericht ein **Tabelle**-Element hinzu. Dann wechseln wir wieder zum Berichtsdaten-Bereich und ziehen einige Felder in die Tabelle, sodass diese wie in Bild 3 aussieht.

Ein Wechsel in die Vorschauansicht liefert eine Liste der einzelnen Bestellpositionen (siehe Bild 4). Damit haben wir alles zusammen, um die angezeigten Datensätze nach verschiedenen Kriterien mit noch zu ermittelnden Parameterwerten zu filtern.

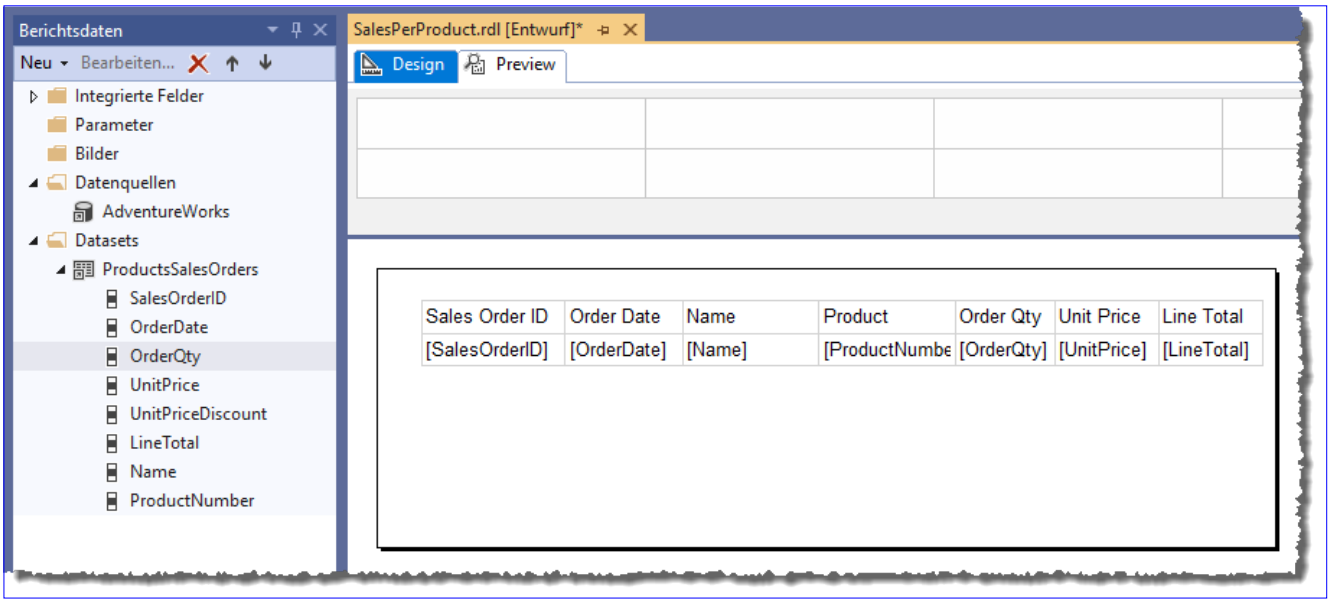
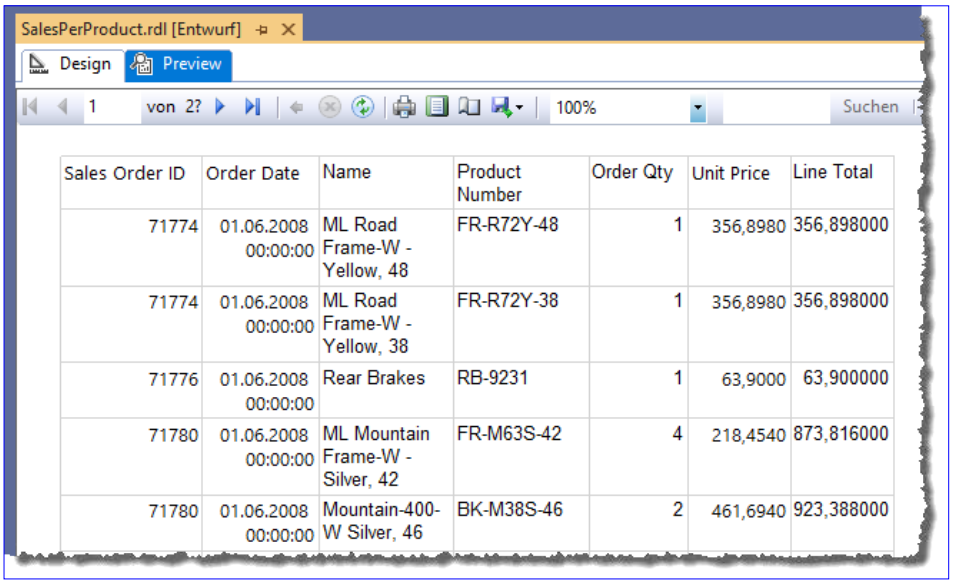


Bild 3: Berichtsentwurf

Parameter zum Dataset hinzufügen

Der erste Schritt ist das Hinzufügen des Parameters zum Dataset. Dazu öffnen wir das Dataset per Doppelklick auf seinen Eintrag im Bereich Berichtsdaten.



Sales Order ID	Order Date	Name	Product Number	Order Qty	Unit Price	Line Total
71774	01.06.2008 00:00:00	ML Road Frame-W - Yellow, 48	FR-R72Y-48	1	356,8980	356,898000
71774	01.06.2008 00:00:00	ML Road Frame-W - Yellow, 38	FR-R72Y-38	1	356,8980	356,898000
71776	01.06.2008 00:00:00	Rear Brakes	RB-9231	1	63,9000	63,900000
71780	01.06.2008 00:00:00	ML Mountain Frame-W - Silver, 42	FR-M63S-42	4	218,4540	873,816000
71780	01.06.2008 00:00:00	Mountain-400-W Silver, 46	BK-M38S-46	2	461,6940	923,388000

Bild 4: Ausgabe der einzelnen Bestellpositionen

Im Bereich **Abfrage** des Dialogs **Dateseteigenschaften** finden wir im Feld **Abfrage** die SQL-Abfrage. Diese erweitern wir wie in Bild 5 um die folgende **WHERE**-Klausel:

```
WHERE SalesOrderDetail.ProductID = @ProductID
```

Um die Funktionsweise dieses Parameters auszuprobieren, klicken Sie auf die Schaltfläche **Abfrage-Designer...** und öffnen so den Dialog **Abfrage-Designer**. Hier finden Sie die neu hinzugefügte **WHERE**-Klausel ebenfalls vor. Um diese auszuprobieren, betätigen Sie die Schaltfläche mit dem roten Ausrufezeichen. Dadurch erscheint der Dialog **Abfrageparameter** (siehe Bild 6). Hier geben Sie den Wert ein, nach dem das Feld **ProductID** gefiltert werden soll, zum Beispiel **991**.

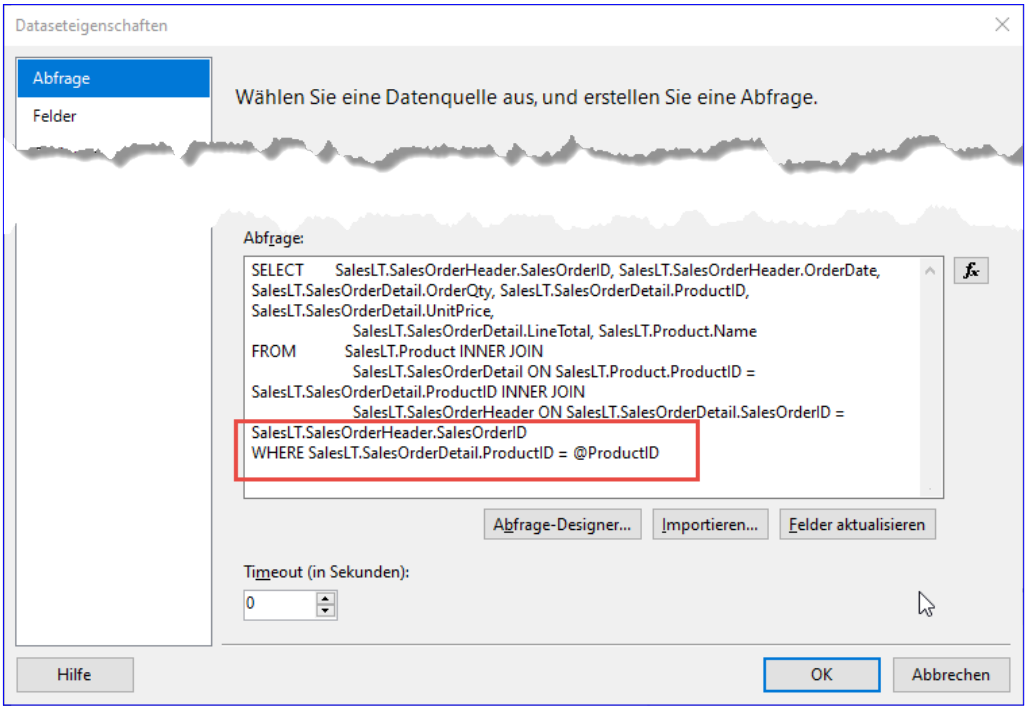


Bild 5: Integrieren des Parameters im Abfrageausdruck

Der Abfrage-Designer zeigt dann im unteren Bereich alle passenden Datensätze an. Nun wollen wir dafür sorgen, dass der Para-

meter auch beim Anzeigen des Berichts abgefragt und berücksichtigt wird.

Vorher werfen wir jedoch noch einen Blick auf den Bereich **Berichtsdaten**. Hier finden wir nämlich nun einen neuen Eintrag namens **[@] ProductID** vor (siehe Bild 7).

Eigenschaften des neuen Parameters festlegen

Den neuen Parameter wollen wir direkt anpassen. Die Eigenschaften des neuen Parameters zeigen Sie an, indem Sie doppelt auf den Eintrag im Ordner **Parameter** klicken. Dies öffnet den Dialog **Berichtsparametereigenschaften** (siehe Bild 8).

Der erste Bereich namens **Allgemein** erwartet die folgenden Informationen:

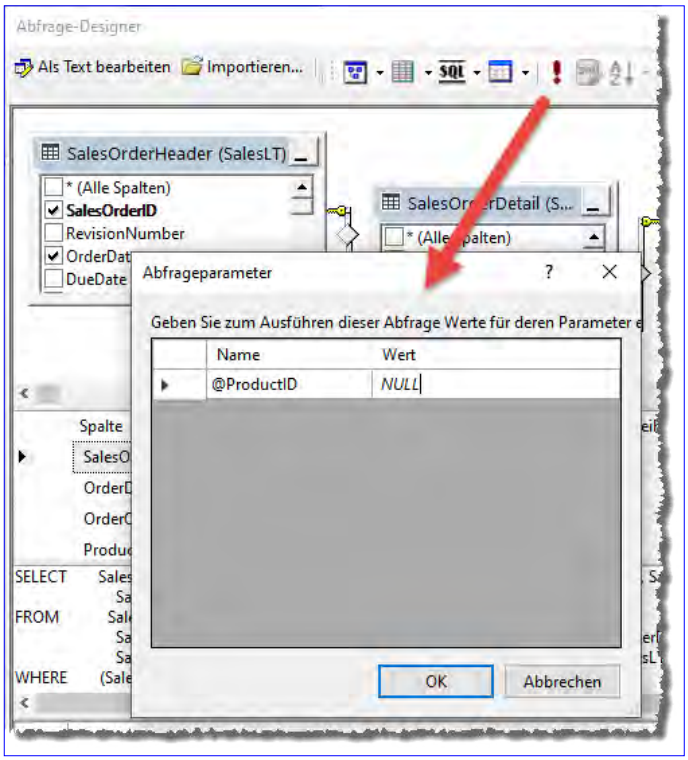


Bild 6: Eingabe des Parameterwertes

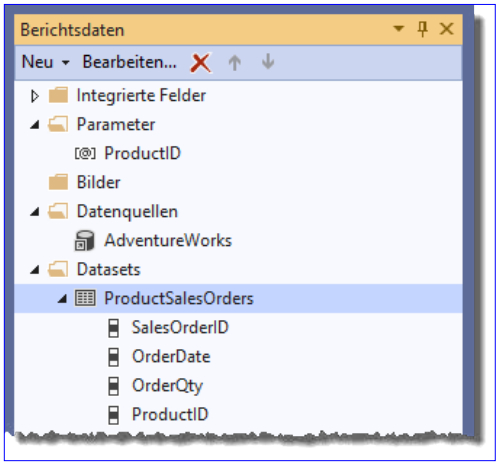


Bild 7: Neuer Eintrag im Ordner **Parameter**

- **Name**: Bezeichnung des Parameters, unter dem Sie diesen später referenzieren können. Dieser wurde bereits mit **ProductID** gefüllt.
- **Eingabeaufforderung**: Text, der erscheint, um den Wert des Parameters vom Benutzer abzufragen. Den vorausgefüllten Text ändern wir in **Filtern nach Produkt-ID:**.
- **Datentyp**: Datentyp des Parameters. Mögliche Werte sind **Text**, **Boolean**, **Datum/Uhrzeit**, **Ganze Zahl** oder **Gleitkommawert**. Wenn Sie **Datum/Uhrzeit** auswählen, bekommt der Benutzer ein entsprechendes Steuerelement zur Auswahl des Datums angezeigt. In diesem Fall benötigen wir den Wert **Ganze Zahl**.
- **Leeren Wert zulassen**: Gibt an, ob leere Zeichenketten angegeben werden können.
- **NULL-Wert zulassen**: Gibt an, ob NULL-Werte zulässig sind. Dies ist nötig, wenn Sie wie weiter unten beschrieben alle Datensätze des Dataset anzeigen wollen.
- **Mehrere Werte zulassen**: Gibt an, ob die Eingabe mehrerer Werte möglich sein soll. Dafür darf die vorherige Eigenschaft keine NULL-Werte zulassen. Später zeigen wir ein Beispiel für diese Einstellung.

Anzeige eines Textfeldes zur Parametereingabe

Neben dem neuen Eintrag im Ordner Parameter sehen wir außerdem nun, wozu die weißen Felder oben im Berichtsentwurf nützlich sind. Hier zeigen die Reporting Services nämlich die Parametertexte und die Steuerelemente zur Eingabe der Parameter an – in unserem Fall ein Feld mit der Beschriftung **Filtern nach Produkt-ID:**.

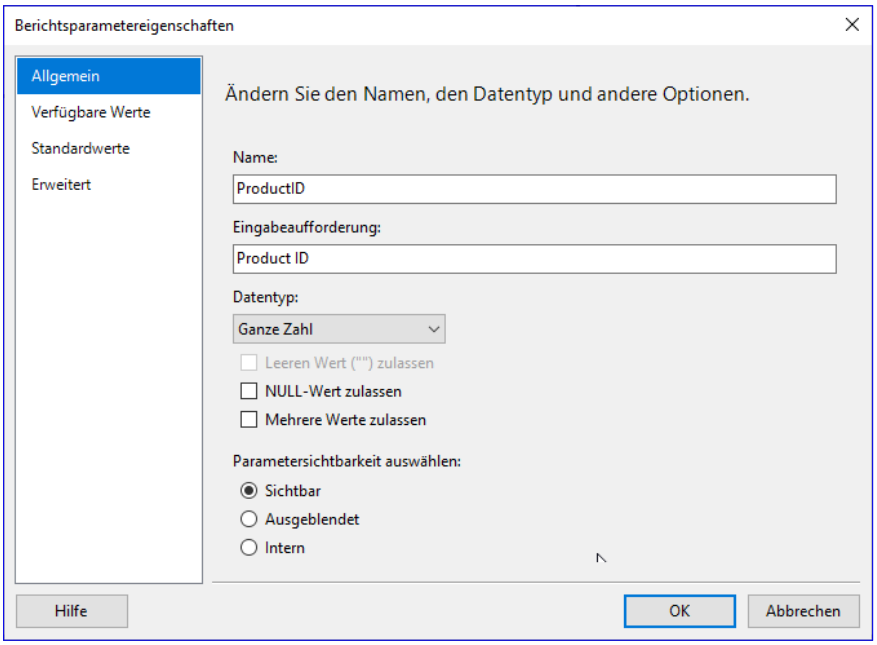


Bild 8: Eigenschaften des neuen Parameters

Wechseln wir nun in die Vorschau-Ansicht, bleibt dieses Eingabefeld erhalten und wir können nach der gewünschten **Produkt-ID** filtern (siehe Bild 9).

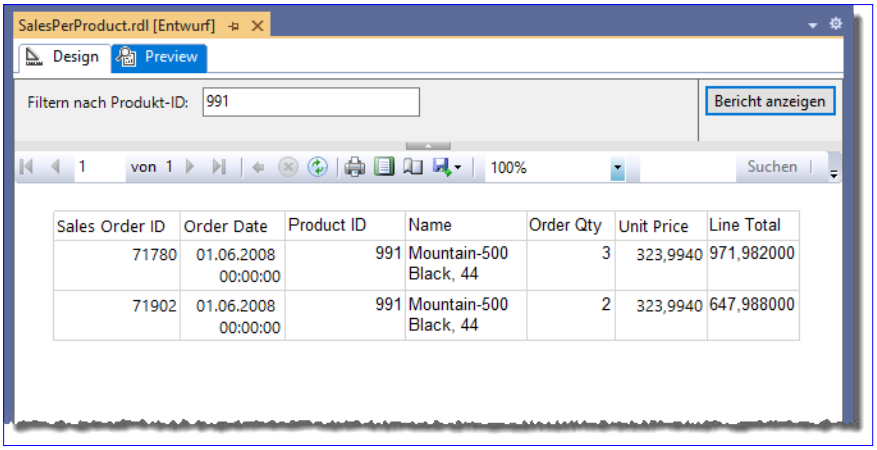


Bild 9: Eingeben des Parameterwertes und Filtern nach der Produkt-ID

Das sieht doch schon sehr benutzerfreundlich aus und viel schöner als beispielsweise die Parameter-Dialoge von Access, in denen man jeden Parameter einzeln eingeben muss. Hier können Sie den Parameterwert ändern und mit der Schaltfläche **Bericht anzeigen** oder einfach durch Betätigen der Eingabetaste die Anzeige aktualisieren.

Alle Datensätze anzeigen, wenn der Parameter NULL ist

Vielleicht möchten Sie, bevor Sie nach einem bestimmten Produkt filtern, erst einmal alle Datensätze des Datasets sehen. In diesem Fall müssen Sie zunächst die Abfrage des Datasets ändern. Die Änderung ist überschaubar – wir fügen lediglich ein weiteres Kriterium zum **WHERE**-Teil hinzu:

```
WHERE (SalesLT.SalesOrderDetail.ProductID = @ProductID OR @ProductID IS NULL)
```

Mit diesem prüfen wir, ob der Parameter **@ProductID** den Wert **NULL** aufweist. Sprich: Wenn der Benutzer keinen Parameter eingegeben hat, liefert die Abfrage alle Datensätze. Das können Sie auch gleich prüfen, indem Sie auf die Schaltfläche **Abfrage-Designer** klicken und im Abfrage-Designer die Abfrage mit einem Klick auf die

Reporting Services: Tabellarische Berichte

Eine der einfachsten Arten, Daten in Berichten darzustellen, ist die tabellarische Darstellung. Die Reporting Services bieten beispielsweise gegenüber Microsoft Access eine erhebliche Vereinfachung: Sie können hier ein Steuerelement speziell zur Anzeige von Daten als Tabelle verwenden. Wie Sie dieses Steuerelement nutzen, und wie Sie auch mehrseitige Berichte mit allen notwendigen Informationen darstellen, zeigt der vorliegende Artikel.

Vorbereitung

An dieser Stelle gehen wir davon aus, dass Sie wie im Artikel **SQL Server Report Builder** (**www.datenbankentwickler.net/258**) den SQL Server Report Builder gestartet haben und dass sowohl eine Datenquelle als auch ein Dataset mit den abzubildenden Daten vorliegt (siehe Bild 1).

Die beiden standardmäßig in einem neuen Bericht enthaltenen Textfelder zum Hinzufügen eines Titels und der Ausführungszeit entfernen wir, indem wir diese markieren und die **Entf**-Taste betätigen. Dadurch erkennen wir, dass der Bericht eigentlich aus einem Berichts- und einem Fußzeile-Bereich besteht (siehe Bild 2).

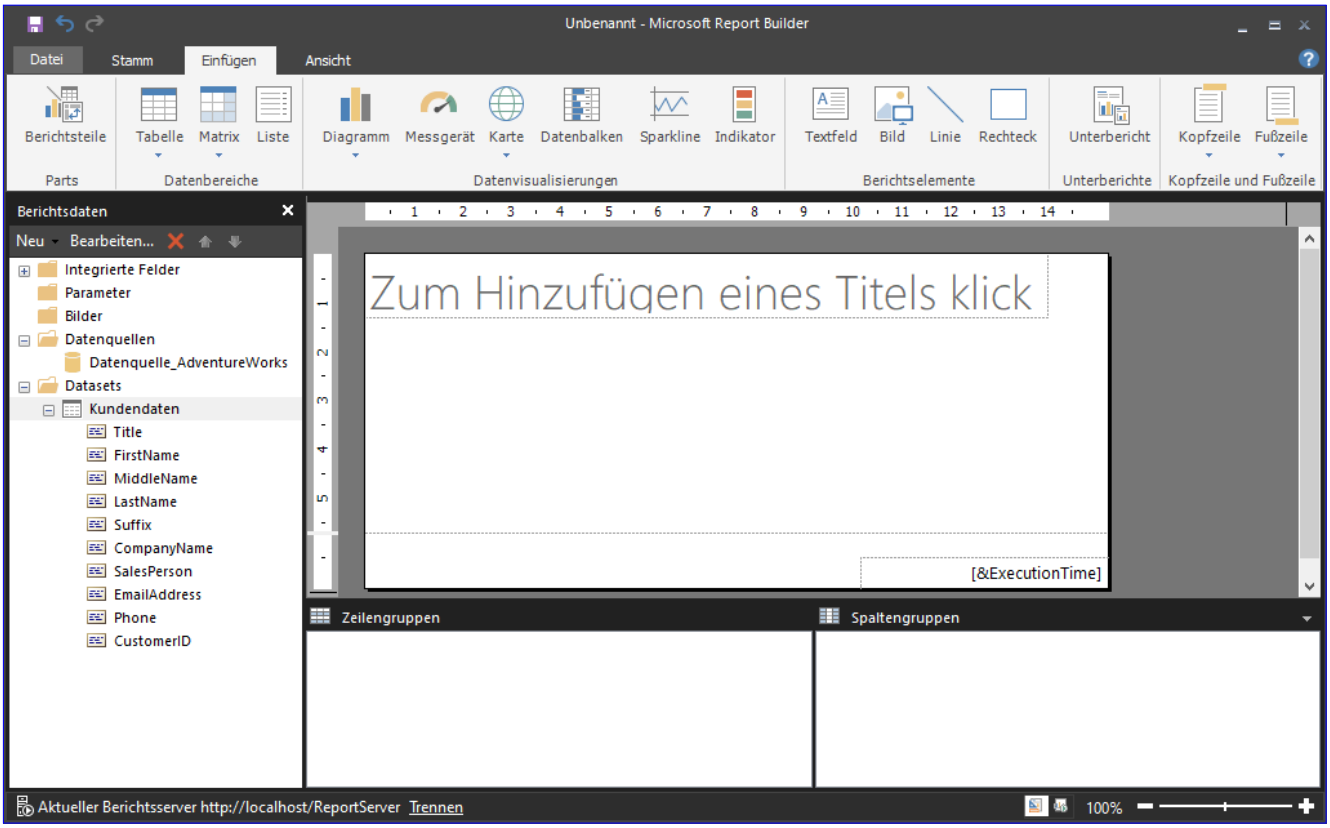


Bild 1: Ausgangssituation für die im vorliegenden Artikel beschriebenen Techniken

Eigenschaften einblenden

Wenn Sie Eigenschaften des aktuell markierten Elements im Berichtsentwurf jederzeit im Überblick haben wollen, wechseln Sie im Ribbon zum Bereich **Ansicht** und aktivieren dort die Option **Eigenschaften**. Hier können Sie außerdem noch den Bereich **Parameter** aktivieren (siehe Bild 3).

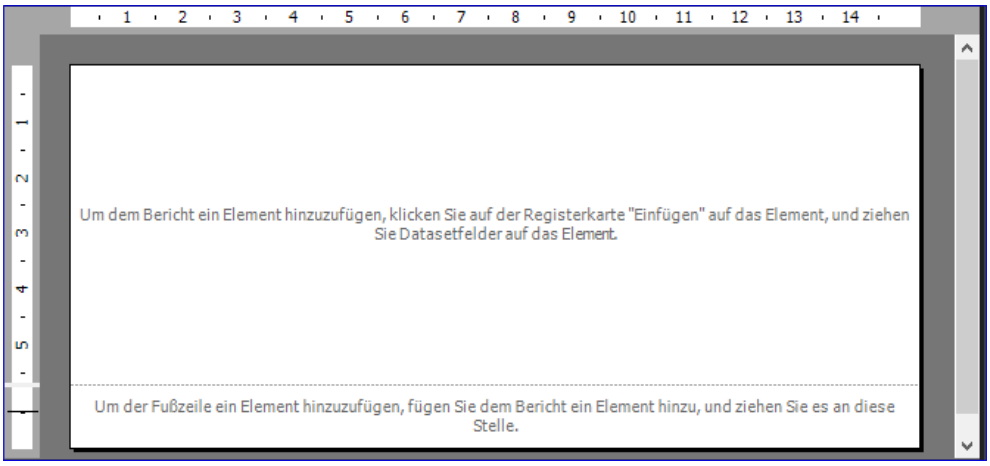


Bild 2: Entwurf des komplett geleerten Berichts

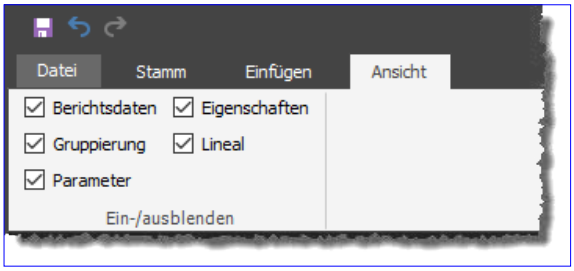


Bild 3: Bereiche ein- und ausblenden

Tabelle hinzufügen

Wir wollen erst einmal eine einfache Tabelle hinzufügen und uns ansehen, welche Möglichkeiten uns diese bietet. Steuerelemente fügen Sie auf folgende Arten hinzu:

- Indem Sie im Ribbon-Bereich **Einfügen** den Eintrag **Tabelle|Tabelle einfügen** anklicken und dann im Entwurf entweder auf den linken oberen Punkt der Tabelle klicken oder einen Rahmen mit der gewünschten Position und Größe aufziehen.

- Die zweite Möglichkeit ist ein rechter Mausklick in den Entwurf und die anschließende Auswahl des einzufügenden Steuerelements, hier also **Einsetzen|Tabelle** (siehe Bild 4).

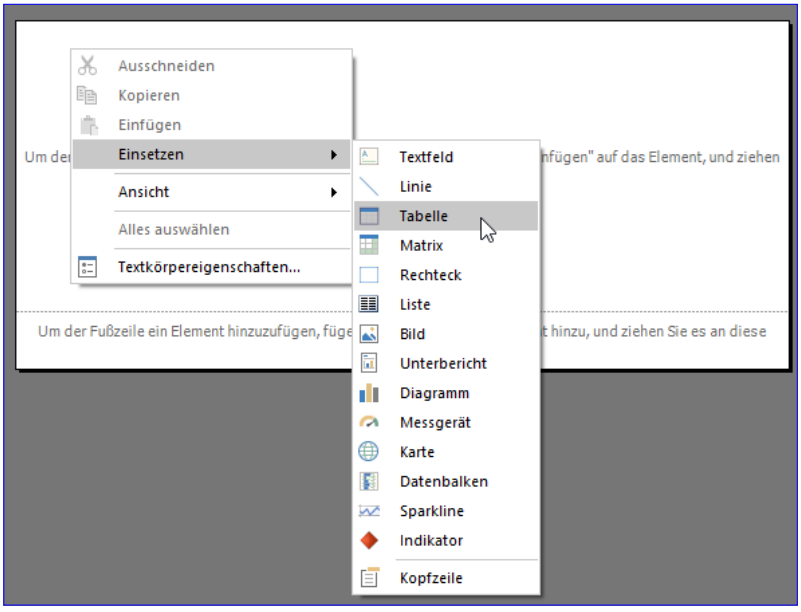


Bild 4: Hinzufügen einer Tabelle per Kontextmenü

Das Ergebnis ist eine neue Tabelle mit zwei Zeilen und drei Spalten, die nach dem Anklicken noch einige graue Kästen links und über den eigentlichen Zellen anzeigt (siehe Bild 5).

Felder hinzufügen per Drag and Drop

Sie können nun die gewünschten Felder aus dem Bereich Berichtsdaten direkt in

die gewünschten Spalten der Tabelle ziehen. Um ein Feld in eine Spalte zu ziehen, bewegen Sie dieses mit der Maus genau in die beiden übereinander angeordneten Felder, welche die Spalte ausmachen. Wenn das Plus-Zeichen im Mauszeiger erscheint, können Sie das Feld fallen lassen.

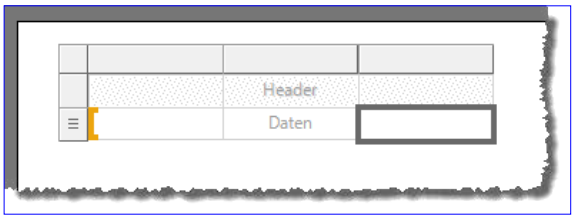


Bild 5: Eine neue, leere Tabelle

Die drei Spalten sind jedoch schnell voll. Was dann? Dann ziehen Sie weitere Felder einfach auf die schmalen Linien zwischen den Spalten beziehungsweise vor die erste oder hinter die letzte Spalte und lassen das Feld dann fallen. Auf diese Weise können Sie beliebig viele Felder hinzufügen, ohne zwischendurch mit anderen Methoden neue Spalten hinzufügen zu müssen.

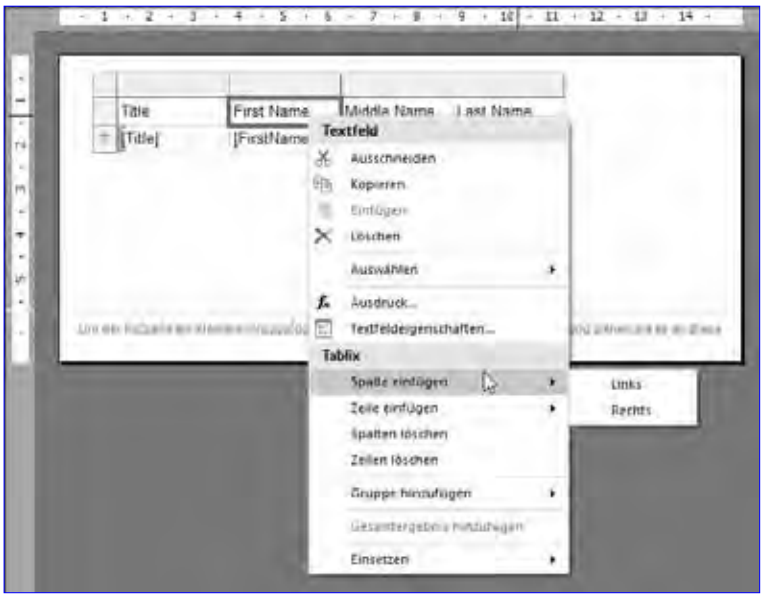


Bild 6: Einfügen neuer Spalten

Falls Sie dennoch einmal weitere Spalten benötigen, eine Spalte löschen wollen oder andere Operationen durchführen wollen, können Sie dazu das Kontextmenü einer Spalte nutzen (siehe Bild 6).

Berichtsabmessungen

Auf diese Weise können Sie alle benötigten Felder zur Tabelle hinzufügen. Irgendwann reicht dann vielleicht der Platz in der Breite nicht mehr und Sie machen den weiß hinterlegten Bereich breiter. Hier gilt es dann zu beachten, welches Format für den Bericht ausgewählt ist und welche Seitenränder für diesen hinterlegt sind.

Die Informationen, die Sie nun benötigen, finden Sie im Bereich Eigenschaften, wenn Sie den Hintergrund des Berichts anklicken (siehe Bild 7). Hier finden Sie unter Seite zunächst die Eigenschaft **PageSize**. Dies entspricht in unserem Fall den Abmessungen einer DIN A4-Seite. Darunter finden Sie in der Eigenschaft **Margins** die vorgesehenen Seitenränder, in diesem Fall **2cm** an jeder Seite. Für die Breite des Berichts bedeutet das, dass Sie **21cm** minus zwei Mal **2cm** zur Verfügung haben, also **17cm**.

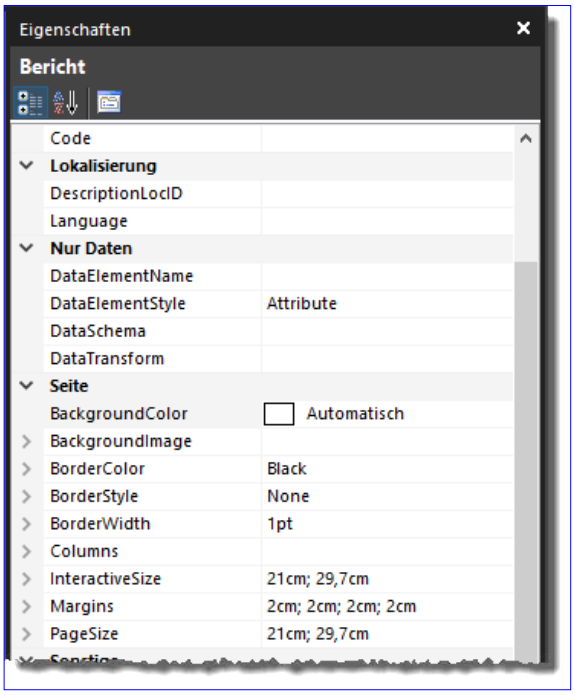


Bild 7: Abmessungen des Berichts und Seitenränder

Die Elemente sollten in der Breite also nicht mehr als **17cm** einnehmen, da sonst Teile der Tabelle auf einer weiteren Seite ausgegeben werden.

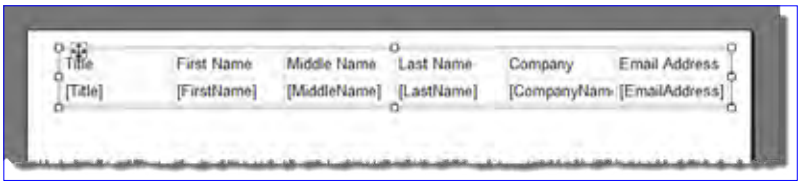


Bild 8: Verschieben einer Tabelle

Tabelle verschieben

Wenn Ihnen die Position der Tabelle noch nicht gefällt, können Sie diese verschieben. Das können Sie erledigen, sobald wie in Bild 8 oben links die Markierung zum Verschieben von Objekten erscheint. Das erreichen Sie beispielsweise folgendermaßen:

- Ziehen Sie einen Rahmen auf, der die Tabelle teilweise oder komplett erfasst.
- Klicken Sie in die Tabelle und betätigen dann die **Esc**-Taste.

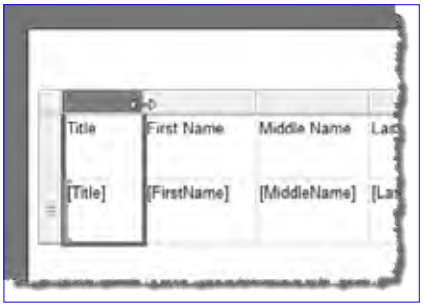


Bild 9: Einstellen der Spaltenbreite

Danach können Sie die Tabelle irgendwo am Rand mit der Maus greifen und an die gewünschte Stelle verschieben.

Größe der Tabelle ändern

Um die Größe einer Tabelle zu ändern, markieren Sie diese wie oben zum Verschieben beschrieben. Danach können Sie die Größe durch Ziehen an einem der Vierecke am Rand der Tabelle einstellen. Die Tabellenfelder passen ihre Höhe und Breite proportional zu den neuen Abmessungen an.

Zeilenhöhe und Spaltenbreite anpassen

Wenn Sie die Zeilenhöhe anpassen wollen, klicken Sie einmal in die Tabelle, damit die grauen Kästchen als Markierung der Spalten und Zeilen erscheinen. Dann können Sie den kleinen grauen Strich zwischen zwei Zeilen zwischen den grauen Kästchen verschieben und so die Höhe der jeweils darüber liegenden Zeile anpassen. Auf die gleiche Weise funktioniert es für die Spalten. Ziehen Sie den Strich zwischen zwei grauen Kästen über den betroffenen Spalten nach links oder rechts, um die Breite der links davon befindliche Spalte einzustellen (siehe Bild 9).

Beachten Sie: Die Breite der rechts von der angepassten Spalte bleibt unverändert. Wenn Sie die Breite vergrößern, werden die rechts davon liegenden Spalten nach rechts verschoben. Gleiches gilt für die Zeilenhöhen.

Spaltenbeschriftungen anpassen

Wenn wir ein Datenmodell in englischer Sprache verwenden, aber eine deutschsprachige Anwendung darauf aufsetzen wollen, können wir mit den englischen Zeilenüberschriften, die automatisch auf Basis der Feldnamen eingefügt wurden, nichts anfangen. Diese können Sie jedoch leicht anpassen. Klicken Sie einmal auf die anzupassende Beschriftung und dann noch einmal oder klicken Sie einmal darauf und geben dann ein Zeichen ein, um den Bearbeitungsmodus zu aktivieren.

Reporting Services: Gruppen und Summen

Im Artikel »Reporting Services: Tabellarische Berichte« haben wir uns am Beispiel einer Kundenliste bereits angesehen, wie Sie einfache Tabellen erstellen und einfache Features wie sich wiederholende Spaltenüberschriften nutzen. In diesem Artikel wollen wir Kunden, Bestellungen und Bestellpositionen zum Thema machen und dazu wiederum eine Tabelle verwenden. Wegen der verschiedenen Ebenen soll diese allerdings Gruppierungen enthalten, die uns unter anderem Informationen über die Umsätze je Position, Bestellung und Kunde liefern.

Vorbereitungen

Als Vorbereitung benötigen Sie die Beispieldatenbank **AdventureWorks**. Deren Download und Installation beschreiben wir im Artikel **AdventureWorks: Schnelle Beispieldatenbank** (**www.datenbankentwickler.net/254**).

Ziel

Wir möchten einen Bericht erstellen, der die Bestellpositionen nach Bestellung und Kunde gruppiert und die Gruppensummen anzeigt. Die Detaildatensätze sollen also jeweils eine Bestellposition enthalten. Die Bestellpositionen einer Bestellung sollen einen Gruppenkopf enthalten, der das Bestelldatum enthält und einen Gruppenfuß mit der Bestellsumme. Außerdem wollen wir noch eine weitere Gruppierung hinzufügen, welche die Bestellungen nach Kunde gruppiert. Diese soll im Gruppenkopf einige wesentliche Kundendaten anzeigen wie die Kundennummer, Vor- und Nachname und die Firma. Im Gruppenfuß dieser Gruppe soll die Summe der Bestellpositionen erscheinen, also der gesamte Umsatz mit diesem Kunden.

Umsatz je Kunde anzeigen

Wir nähern uns dem Schritt für Schritt. Zunächst möchten wir die Umsatzsummen je Kunde in jeweils einer Zeile des Berichts anzeigen. Jede Zeile der Tabelle soll also einen Kunden und die Umsätze dieses Kunden liefern.

Gruppierungen per Assistent

Im ersten Anlauf wollen wir die gruppierte Tabelle per Assistent erstellen. Dazu rufen wir im SQL Server Report Builder mit dem Ribbonbefehl **Datei|Neu** den Dialog **Neuer Bericht oder neues Dataset** auf. Hier wählen wir den Eintrag **Tabellen- oder Matrix-Assistent** (siehe Bild 1).

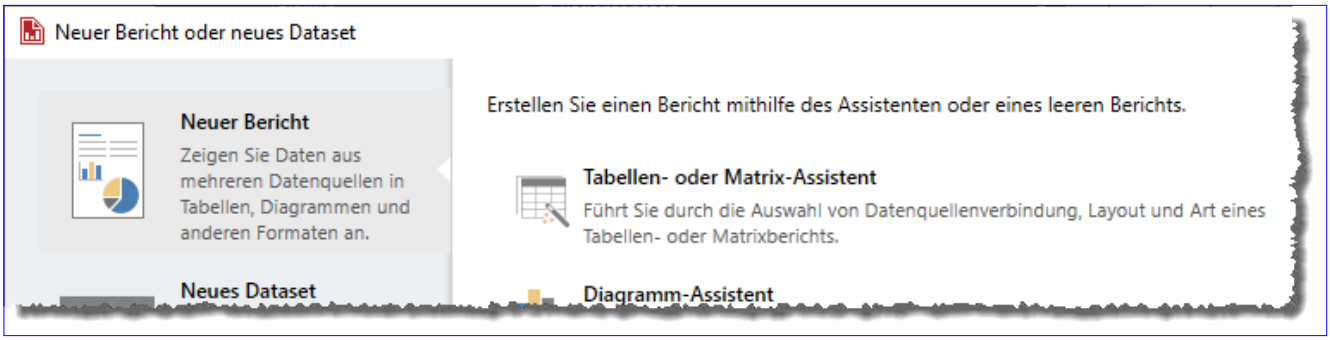


Bild 1: Starten des Assistenten

Datenquelle festlegen

Im nun folgenden Dialog **Neue Tabelle oder Matrix** aktivieren wir die Option **Dataset erstellen** und klicken auf **Weiter**. Damit gelangen wir zum Schritt **Verbindung mit einer Datenquelle auswählen**. Hier klicken Sie auf die Schaltfläche **Neu**. Im Dialog **Datenquelleneigenschaften** definieren Sie die Datenquelle, die beispielsweise wie in Bild 2 aussieht.

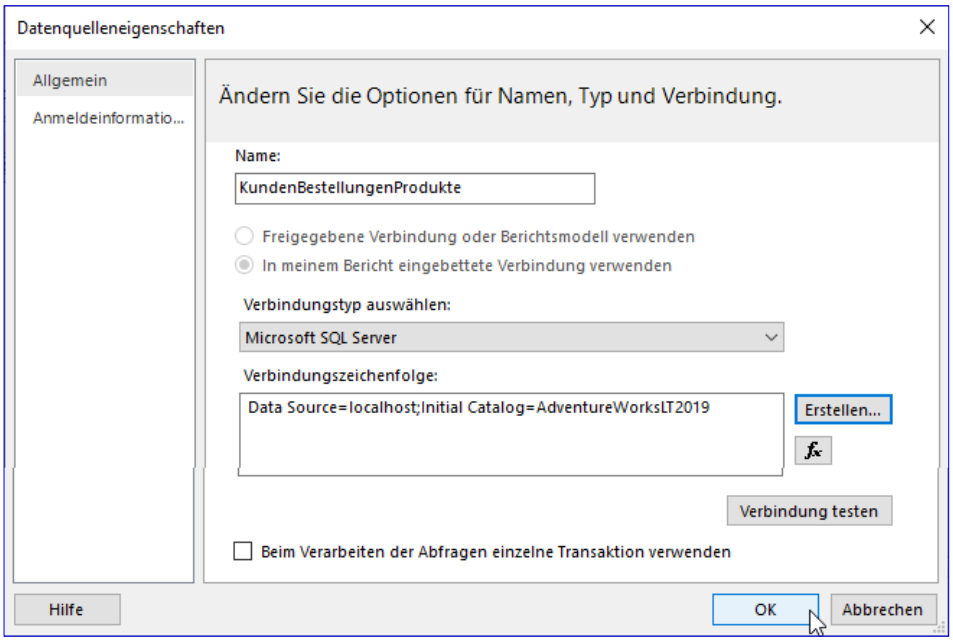


Bild 2: Datenquelle definieren

Dataset definieren

Zurück im Assistenten sehen Sie den Schritt Abfrage entwerfen. Hier wählen Sie im linken Bereich die kompletten Tabellen oder auch nur einzelne Felder aus, die im Bericht verwendet werden sollen.

Aus Gründen der Übersicht sollten Sie nur die Felder wählen, die Sie später auch nutzen. In diesem Fall nutzen wir Felder der Tabellen **Customer**, **Product**, **ProductCategory**, **SalesOrderDetail** und **SalesOrderHeader**. Mit einem Klick auf die Schaltfläche **Abfrage ausführen** können Sie sich die zu erwartenden

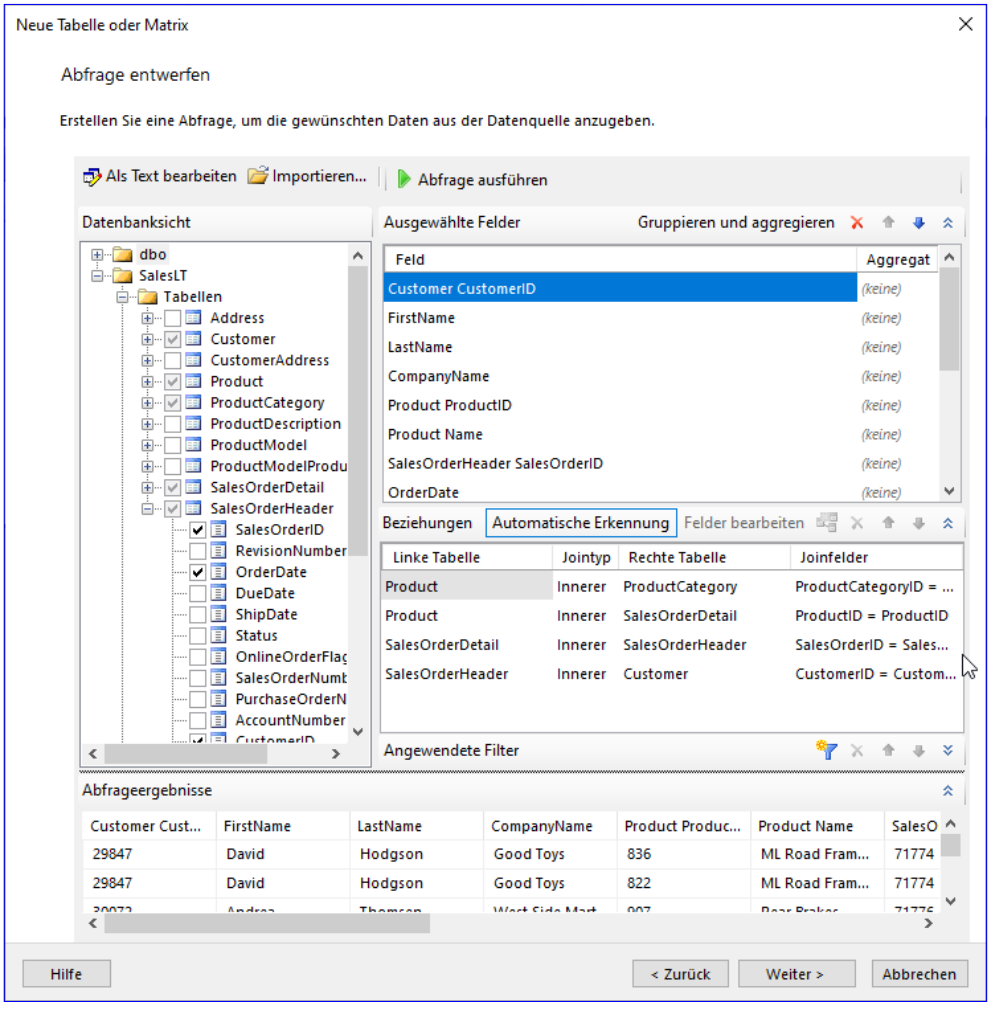


Bild 3: Dataset definieren

Daten direkt als Abfrageergebnis anschauen (siehe Bild 3).

Wie Sie Datenquelle und Dataset ohne Assistent anlegen, beschreiben wir im Detail Artikel **SQL Server Report Builder** (**www.datenbankentwickler.net/258**).

Gruppierungen definieren

Nachdem wir festgelegt haben, welche Daten wir im Bericht sehen wollen, folgt im nächsten Schritt die Anordnung der Felder. Hier wird es interessant, denn hier finden wir vier Listen vor:

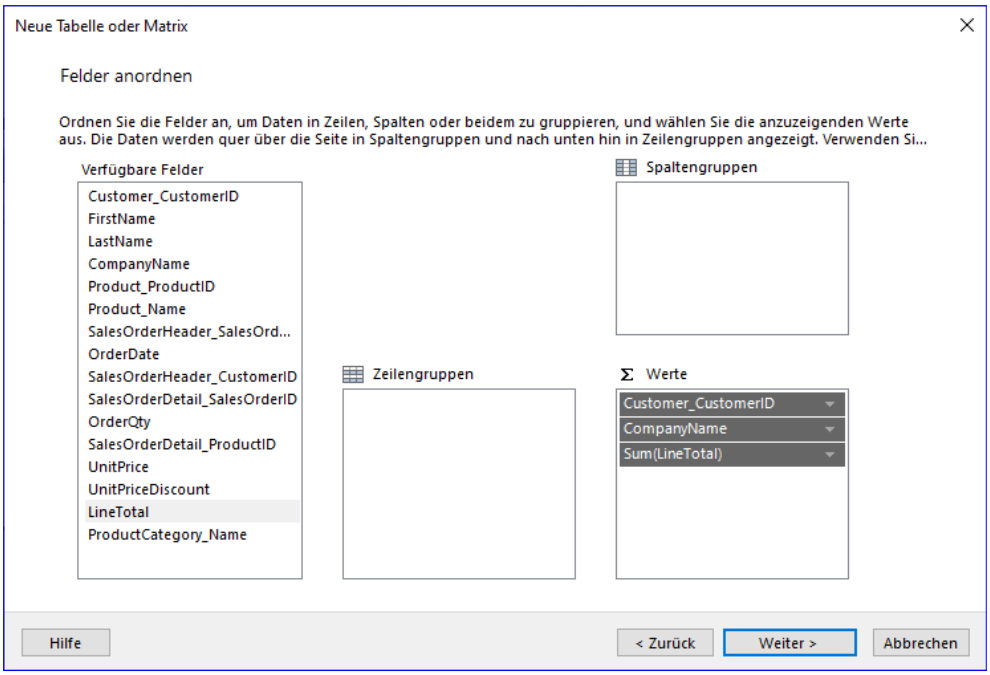


Bild 4: Felder für die Tabelle auswählen

- **Verfügbare Felder**: Alle Felder, die wir im vorherigen Schritt selektiert haben.
- **Spaltengruppen**:
- **Zeilengruppen**:
- **Werte**:

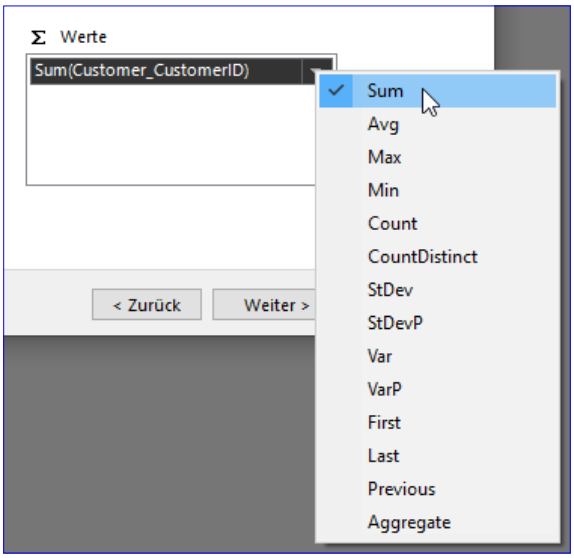


Bild 5: Einstellen der Aggregatfunktion

Im ersten Beispiel wollen wir nur die Umsätze nach Kunden ausgeben. Dazu reicht es aus, wenn wir einfach die anzuzeigenden Felder in den Bereich **Werte** ziehen. Wir wählen hier die Kundennummer (**CustomerID**), die Firmenbezeichnung (**CustomerName**) und den Umsatz je Bestellposition, also das Feld **LineTotal** der Tabelle **SalesOrderDetail** (siehe Bild 4). Für Felder mit numerischen Datentypen stellt der Assistent automatisch die **Sum**-Funktion als Aggregatfunktion ein. Für das Feld **CustomerID** wollen wir das jedoch aus nachvollziehbaren Gründen nicht.

Um dies zu ändern, klicken Sie einfach auf die Schaltfläche mit dem nach unten zeigenden Dreieck. Es erscheint eine Liste aller Aggregatfunktionen. Vielleicht suchen Sie wie in Access-Abfragen nach einem Eintrag wie Ausdruck. Diesen gibt es nicht: Damit einfach nur der Wert des Feldes angezeigt wird, klicken Sie einfach auf die

Zeile, die mit dem Haken markiert ist. Dann ist keine Aggregatfunktion mehr für dieses Feld festgelegt (siehe Bild 5). Wenn Sie das Textfeld **CompanyName** hinzufügen, wird automatisch keine Aggregatfunktion selektiert. Erst für das Feld **LineTotal** legt der Assistent wieder die Aggregatfunktion **Sum** fest, die in diesem Fall ja auch erwünscht ist.

Die folgenden beiden Schritte des Assistenten bieten aufgrund der aktuellen Einstellungen keine weiteren Auswahlmöglichkeiten, sodass wir die Erstellung abschließen können. Das Ergebnis der Arbeit mit dem Assistenten sieht wie in Bild 6 aus. Die Spaltenbreiten haben wir nachträglich vergrößert.

Wenn wir den Bericht nun ausführen und in die **Seitenlayout**-Ansicht wechseln, erhalten wir den Bericht mit dem Ausschnitt aus Bild 7. Die angezeigten Daten entsprechen unseren Wünschen, allein am Layout könnte man noch arbeiten. Beispielsweise sollen die Spaltenüberschriften deutsche Texte erhalten und der Umsatz soll in der Währung **EUR** angezeigt werden.

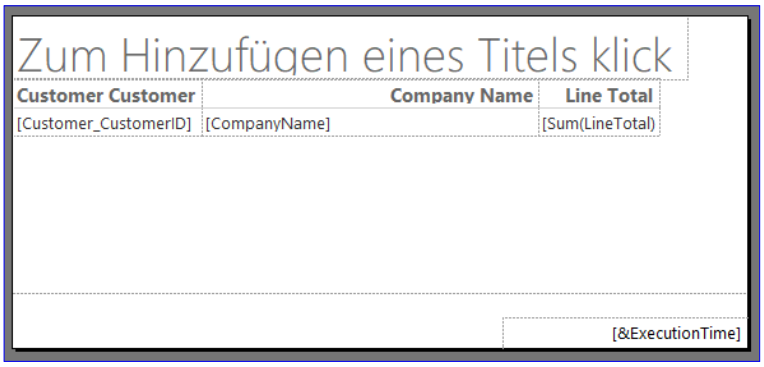
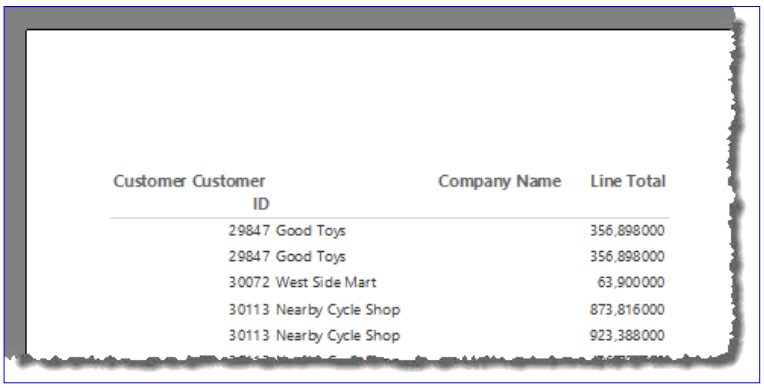
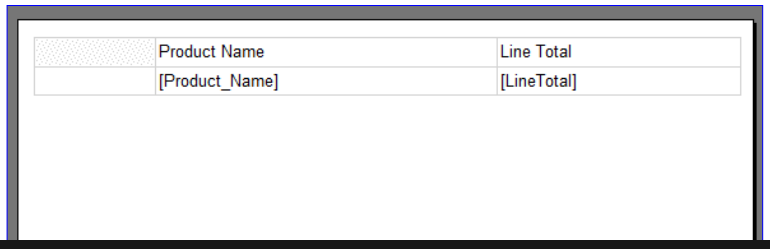


Bild 6: Resultierender Entwurf des Assistenten



Customer Customer ID	Company Name	Line Total
29847	Good Toys	356,898000
29847	Good Toys	356,898000
30072	West Side Mart	63,900000
30113	Nearby Cycle Shop	873,816000
30113	Nearby Cycle Shop	923,388000

Bild 7: **Seitenlayout**-Ansicht des Berichts



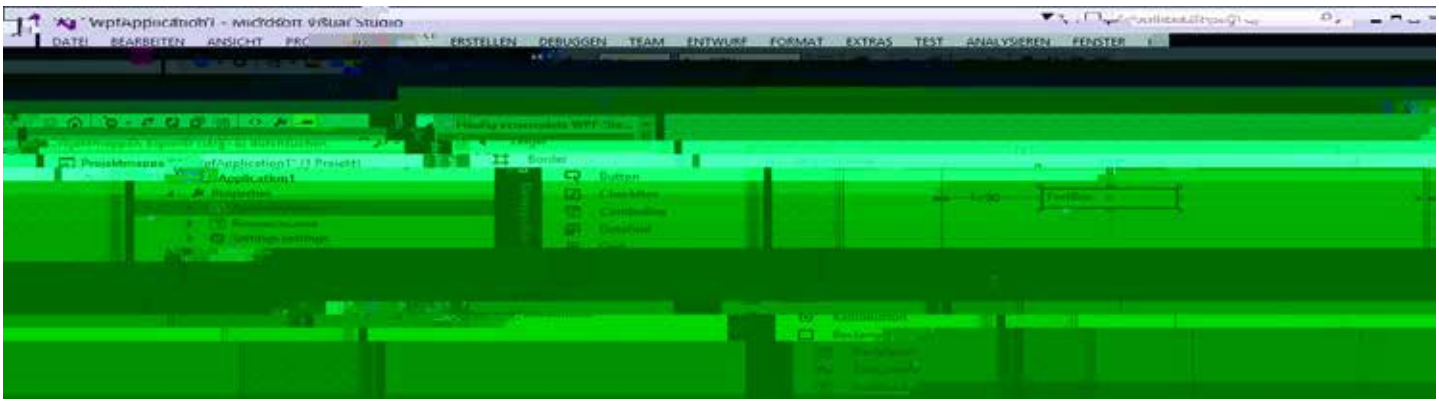
	Product Name	Line Total
	[Product_Name]	[LineTotal]

Ebene für die einzelnen

AUSGABE 03/2021

DATENBANK ENTWICKLER

MAGAZIN FÜR DIE DATENBANKENTWICKLUNG MIT VISUAL STUDIO FÜR DESKTOP, WEB UND CO.



TOP-THEMEN:

STEUERELEMENTE	Das Webbrowser-Steuerelement	SEITE 3
STEUERELEMENTE	Navigieren im Frame-Steuerelement	SEITE 11
VB-GRUNDLAGEN	Generische Auflistungen unter VB.NET	SEITE 30
INTERAKTIV	CSV-Datei in Klassen importieren	SEITE 46

André Minhorst Verlag

Das WPF Webbrowser-Steuerelement

Es gibt verschiedene Gründe, warum Sie ein Webbrowser-Steuerelement in Ihrer Anwendung benötigen könnten. Vielleicht möchten Sie einfach die Möglichkeit bieten, bestimmte Webseiten anzuzeigen. Oder Sie wollen es zur Anzeige von HTML-Inhalten verwenden, die Sie aus Dateien lesen oder per Code zusammenstellen. Schließlich bietet HTML einige Möglichkeiten, die anders oder besser zu realisieren sind als unter WPF. Dann können Sie das Webbrowser-Steuerelement verwenden, um die Benutzeroberfläche der WPF-Anwendung zu erweitern. Dieser Artikel erläutert die Grundlagen für die Programmierung des WPF Webbrowser-Steuerelements.

WPF Webbrowser-Steuerelement hinzufügen

Das WPF Webbrowser-Steuerelement fügen Sie wie die übrigen Steuerelemente auf verschiedene Arten zu einem WPF-Fenster hinzu. Sie können es beispielsweise aus dem Bereich **Alle WPF-Steuerelemente** aus der **Toolbox** in das WPF-Fenster ziehen oder dieses anklicken und dann im WPF-Fenster einen Rahmen aufziehen, der direkt die Größe des Steuerelements angibt.

Wer lieber mit Code arbeitet, fügt schlicht das Element **Webbrowser** in das **Grid**-Element des Fensters ein. Wir legen hier mit **wb** direkt den Namen des Steuerelements fest, damit wir es später leichter referenzieren können:

```
<Grid>
    <WebBrowser x:Name="wb"></WebBrowser>
</Grid>
```

Damit ist allerdings noch nichts erreicht, denn das WPF Webbrowser-Steuerelement lädt weder von selbst eine Seite noch bietet es Benutzeroberflächen-Elemente wie ein Textfeld zur Eingabe einer URL oder Schaltflächen zum Navigieren. Also legen wir in den nächsten Abschnitten selbst Hand an und fügen die benötigten Elemente hinzu.

Bild 1: Anzeige einer Webseite im WPF Webbrowser-Steuerelement

Anzeigen einer Webseite beim Starten der Anwendung

Damit das WPF Webbrowser-Steuerelement beim Starten der Anwendung wie in Bild 1 direkt

eine Webseite anzeigt, fügen wir eine Ereignismethode hinzu, die durch das Ereignis **Loaded** des **Window**-Elements ausgelöst wird. Das benötigte Attribut sieht wie folgt aus:

```
<Window x:Class="MainWindow" ...  Loaded="Window_Loaded">
```

Die Ereignismethode **Window_Loaded** implementieren wir wie folgt und fügen als einzige Anweisung einen Aufruf der **Navigate**-Methode des WPF Webbrowser-Steuerelements hinzu, das wir mit **wb** benannt haben. Die **Navigate**-Methode erwartet die URL als Parameter:

```
Private Sub Window_Loaded(sender As Object, e As RoutedEventArgs)
    wb.Navigate("http://www.google.de")
End Sub
```

Navigieren per URL-Textfeld

Normalerweise werden Sie nicht nur eine Seite anzeigen wollen, sondern dem Benutzer die Gelegenheit geben wollen, selbst eine URL einzugeben, die das WPF Webbrowser-Steuerelement anzeigen soll. Dazu fügen wir über dem Webbrowser-Steuerelement ein Textfeld plus Bezeichnungsfeld hinzu sowie eine Schaltfläche zum Anzeigen der gewünschten URL.

Im XAML-Code legen wir dafür ein **DockPanel**-Element an, das oben ein **Grid**-Element mit **Label**, **TextBox** und **Button** enthält sowie unten das **Webbrowser**-Steuerelement:

```
<DockPanel>
    <Grid DockPanel.Dock="Top"><
        <Grid.ColumnDefinitions>
            <ColumnDefinition Width="Auto"></ColumnDefinition>
            <ColumnDefinition Width="*"></ColumnDefinition>
            <ColumnDefinition Width="Auto"></ColumnDefinition>
        </Grid.ColumnDefinitions>
        <Label Grid.Column="0">URL:</Label>
        <TextBox x:Name="txtURL" Grid.Column="1"></TextBox>
        <Button x:Name="btnNavigate" Grid.Column="2" Click="btnNavigate_Click">Navigate</Button>
    </Grid>
    <WebBrowser x:Name="wb"></WebBrowser>
</DockPanel>
```

Navigieren per Klick auslösen

Um den Navigationsvorgang auszulösen, reicht eigentlich die folgende Methode:

```
Private Sub btnNavigate_Click(sender As Object, e As RoutedEventArgs)
    Dim strURL As String
```

```
    strURL = txtURL.Text
    wb.Navigate(strURL)
End Sub
```

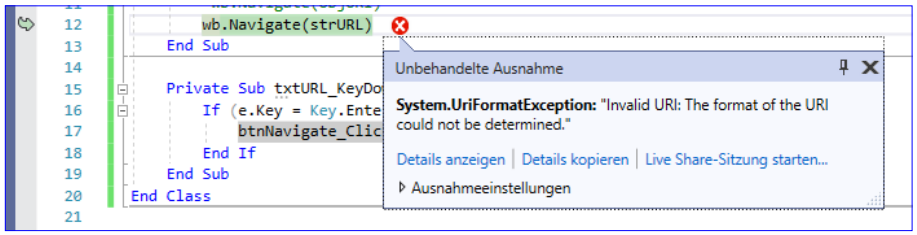


Bild 2: Fehler bei Angabe einer URL ohne **http://**

Diese setzt allerdings voraus, dass der Benutzer eine vollständige URL liefert, also beispielsweise inklusive **http://** oder **https://**. Dann können wir beim Anklicken der Schaltfläche den Text des Textfeldes **txtURL** in eine **String**-Variable einlesen und deren Inhalt der Methode **Navigate** als Parameter übergeben. Gibt der Benutzer allerdings beispielsweise nur **www.google.de** an, erhalten wir den Fehler aus Bild 2.

Wenn wir diesen Fehler ausschließen wollen, können wir eine leicht abgewandelte Version der zuvor vorgestellten Methode nutzen. Hier führen wir das Objekt **objUri** mit dem Typ **Uri** ein. Wir schreiben den Inhalt von **txtURL** wieder in **strURL**. Dann erstellen wir ein neues Objekt des Typs **UriBuilder** und übergeben diesem den Inhalt von **strURL**. Die Eigenschaft **Uri** dieses Objekts liefert das erstellte **Uri**-Objekt und wir speichern es in **objURI**. Dieses übergeben wir schließlich als Parameter an die **Navigate**-Methode:

```
Private Sub btnNavigate_Click(sender As Object, e As RoutedEventArgs)
    Dim strURL As String
    Dim objURI As Uri
    strURL = txtURL.Text
    objURI = New UriBuilder(strURL).Uri
    wb.Navigate(objURI)
End Sub
```

Auf diese Weise kann der Benutzer auch Internetadressen ohne Angabe von **http://** oder **https://** in das Textfeld eingeben.

Seite direkt beim Betätigen der Eingabetaste öffnen

Benutzerfreundlicher wird die Lösung, wenn wir dem Benutzer die Betätigung der Eingabetaste zum Aufrufen der angegebenen Seite anbieten. Dazu fügen wir dem **TextBox**-Element **txtURL** noch das **KeyDown**-Attribut hinzu, für das wir die folgende Ereignismethode hinterlegen:

```
Private Sub txtURL_KeyDown(sender As Object, e As KeyEventArgs)
    If (e.Key = Key.Enter) Then
        btnNavigate_Click(Me, New RoutedEventArgs)
    End If
End Sub
```

Die Methode prüft, ob das mit den **KeyEventArgs** übergebene Zeichen dem Wert **Key.Enter** entspricht, als der Eingabetaste. In diesem Fall ruft die Methode die zuvor definierte Methode **btnNavigate_Click** auf.

Javascript-Fehler im WPF Webbrowser-Steuerelement

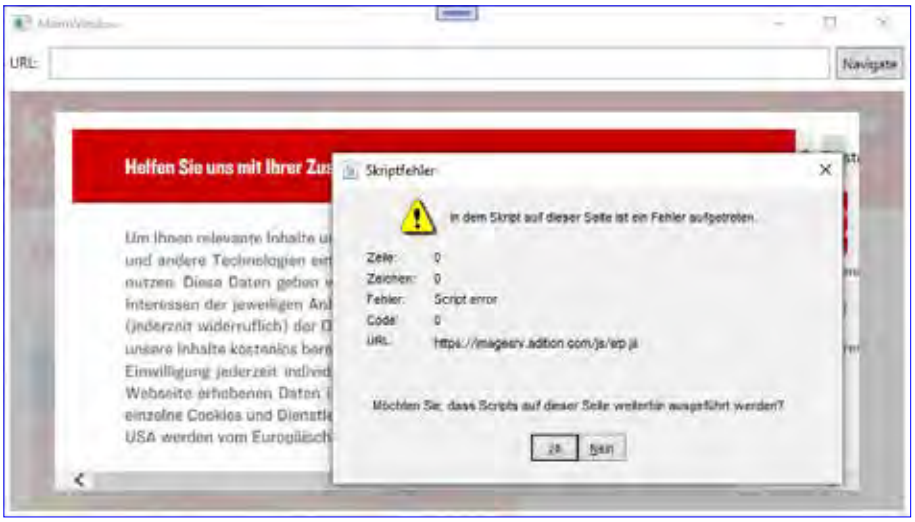


Bild 3: Javascript-Fehler im WPF Webbrowser-Steuerelement

Beim Laden verschiedener Webseiten sind in unserem WPF Webbrowser-Steuerelement durch Javascript ausgelöste Fehler aufgetreten (siehe Bild 3). Warum treten diese hier auf, obwohl sie in anderen Browsern nicht in Erscheinung treten? Das liegt daran, dass das WPF Webbrowser-Steuerelement schlicht nicht so umfangreiche Funktionen wie eigenständige Browser hat. Er ist somit nicht zum Anzeigen aller möglichen Seiten geeignet, vor allem aber nicht von Seiten, die mehr oder weniger Gebrauch von Javascript-Code machen. Sie sollten das WPF Webbrowser-Steuerelement also eher für solche Einsatzzwecke nutzen, wo Sie selbst den anzuzeigenden Inhalt per Code zusammenstellen, der entweder keinen Skript-Code enthält oder wo Sie zumindest die Kontrolle über eventuell auftretende Fehler haben.

Navigationsschaltflächen programmieren

Wenn schon das WPF Webbrowser-Steuerelement verwendet wird, soll der Benutzer auch die Möglichkeit haben, zur vorherigen und zur nachfolgend aufgerufenen Seite zu navigieren. Dazu fügen wir dem Fenster zwei Schaltflächen hinzu, die wir im XAML-Code wie folgt zusammen mit der Schaltfläche **btnNavigate** in einem **StackPanel**-Element definieren:

```
<StackPanel Grid.Column="2" Orientation="Horizontal">
    <Button x:Name="btnNavigate" Grid.Column="2" Click="btnNavigate_Click">Navigate</Button>
    <Button Command="NavigationCommands.BrowseBack">
        <Image Source="images/nav_left.png" Width="16" Height="16" />
    </Button>
    <Button Command="NavigationCommands.BrowseForward">
        <Image Source="images/nav_right.png" Width="16" Height="16" />
    </Button>
</StackPanel>
```

Für die beiden **Button**-Elemente haben wir jeweils ein **Image**-Element angegeben. Die beiden dazu benötigten und im **Source**-Attribut angegebenen Bilddateien speichern wir im Projektmappen-Explorer im Verzeichnis **images**. Die Bilddateien ziehen Sie dazu einfach in das zuvor im Projekt angelegte Verzeichnis hinein. Danach stellen Sie noch sicher, dass die Eigenschaften für die Bilddateien korrekt eingestellt sind. Dazu markieren Sie die

Navigieren im Frame-Steuerelement

Wenn Sie auf einfache Weise verschiedene Seiten in einer WPF-Benutzeroberfläche anzeigen wollen, bietet sich die Nutzung des Frame-Steuerelements zur Anzeige von Page-Elementen an. Das Frame-Steuerelement ist dabei der Rahmen, die Page-Elemente sind im Prinzip Fenster ohne Fensterrahmen. Dieser Artikel zeigt, welche Möglichkeiten das Frame-Element bietet, wie Sie Page-Elemente darin anzeigen und wie die Steuerung der Anzeige verschiedener Page-Elemente funktioniert.

Wenn Sie eine einfache WPF-Anwendung bauen, die ein Ribbon zur Steuerung der Anwendung enthält und einen Bereich, der die zu bearbeitenden Daten anzeigen soll, ist das **Frame**-Steuerelement in Kombination mit **Page**-Elementen für die einzelnen Seiten eine praktische Lösung. Das Ribbon würden Sie dabei in der oberen Zeile eines **Grid**-Elements darstellen, das **Frame**-Element in der zweiten Zeile. Wir wollen im Beispiel nicht direkt mit einem Ribbon arbeiten, sondern mit einfachen Schaltflächen, also fügen wir diese in der ersten Zeile eines Grids ein und das Frame-Element direkt darunter (siehe Bild 1).

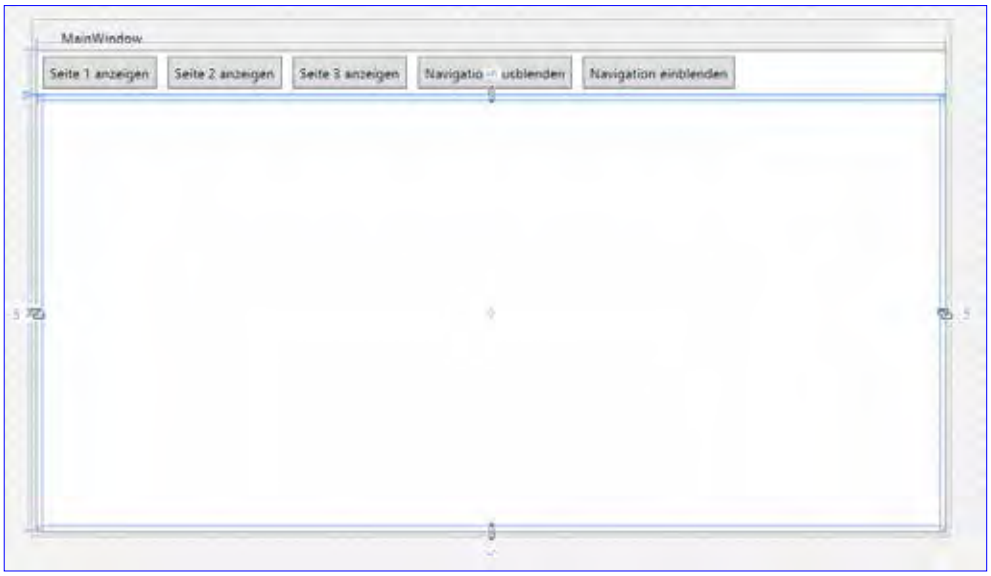


Bild 1: **Frame**-Element mit Schaltflächen

Der Code für die folgenden Beispiele ist wie folgt aufgebaut:

```
<Grid>
    <Grid.RowDefinitions>
        <RowDefinition Height="Auto"></RowDefinition>
        <RowDefinition Height="*"></RowDefinition>
    </Grid.RowDefinitions>
   <StackPanel Grid.Row="0" Orientation="Horizontal">
        <Button x:Name="btnSeite1" Content="Seite 1 anzeigen" Click="btnSeite1_Click"/>
        <Button x:Name="btnSeite2" Content="Seite 2 anzeigen" Click="btnSeite2_Click"/>
        <Button x:Name="btnSeite3" Content="Seite 3 anzeigen" Click="btnSeite3_Click"/>
        ... weitere Steuerelemente
    </StackPanel>
```

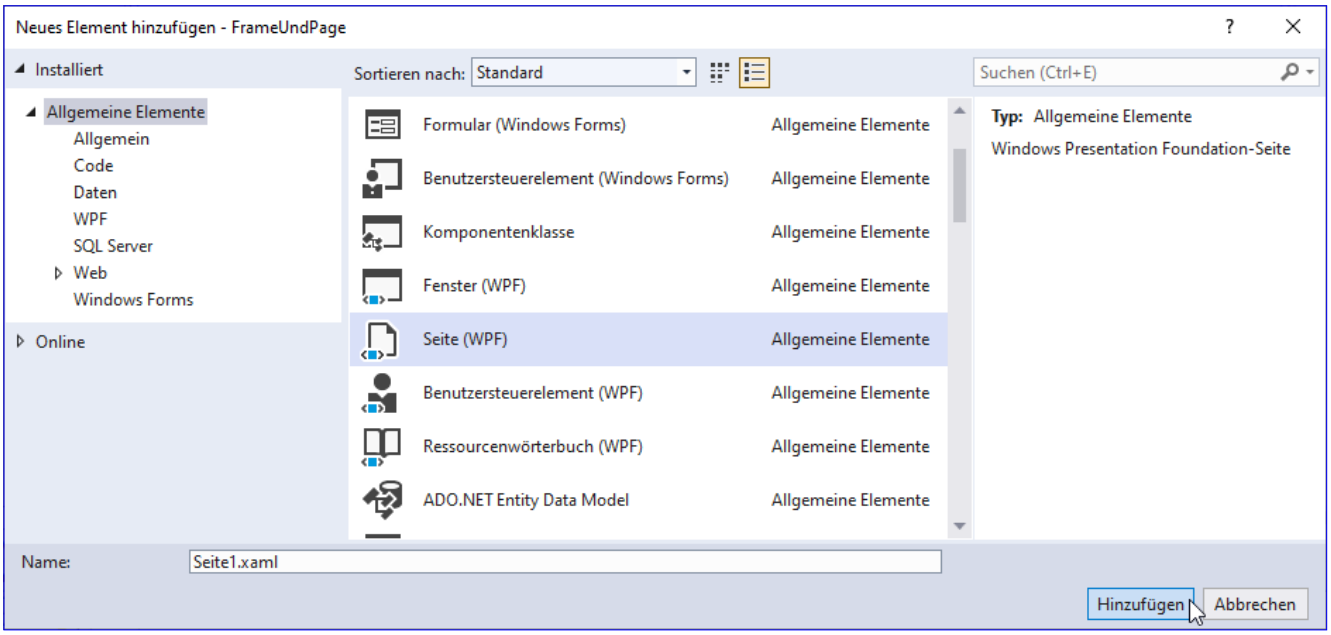


Bild 2: Hinzufügen eines **Page**-Elements

```
    <Frame x:Name="fra" Grid.Row="1"></Frame>
</Grid>
```

Wir definieren hier also ein **Grid** mit zwei Zeilen, von denen die erste die in einem **StackPanel** angeordneten Schaltflächen enthält und die zweite das **Frame**-Steuerelement. Wie kommen nun die Pages ins Spiel? Diese fügen wir zunächst als **Page**-Elemente zum Projekt hinzu – zunächst einmal drei Stück.

Dazu nutzen wir den Befehl **Hinzufügen|Seite (WPF)...** aus dem Kontextmenü des **Projekt**-Elements im Projektmappen-Explorer und geben im nun erscheinenden Dialog **Neues Element hinzufügen** den Namen des hinzuzufügenden **Page**-Elements an – hier **Seite1.xaml** (siehe Bild 2).

Damit wir in der Anwendung erkennen können, welches **Page**-Element wir jeweils anzeigen, fügen wir jedem **Page**-Element ein Label hinzu – beispielsweise mit dem Text **Seite 1**:

```
<Page x:Class="Seite1" ... Title="Seite1">
    <Grid>
        <Label>Seite 1</Label>
    </Grid>
</Page>
```

Im Gegensatz zur Darstellung eines Fensters (**Window**-Elements) mit dem Fensterrahmen und dem weißen Hintergrund erscheint das **Page**-Element ohne ausgeprägten Rahmen und hat auch keinen Hintergrund, das heißt, es ist transparent (siehe Bild 3). Darüber hinaus enthält es jedoch in der XAML-Definition die gleichen Eigenschaften wie ein **Window**-Element. Wenn Sie beispielsweise Ressourcen definieren wollen, um seitenweit

gültige Eigenschaften für Steuerelemente zu definieren, nutzen Sie nicht das Element **Window.Resources**, sondern **Page.Resources**:

```
<Page x:Class="Seite2"
      ...
      Title="Seite2">
    <Page.Resources>
        <Style TargetType="Label">
            <Setter Property="Margin" Value="5"></Setter>
        </Style>
    </Page.Resources>
    ...
```

Page-Element im Frame-Element anzeigen

Wie sorgen wir nun für die Anzeige von **Page**-Elementen im **Frame**-Element?

Dazu können wir verschiedene Techniken nutzen:

- **Content**-Eigenschaft
- **Navigate**-Methode
- Hyperlinks
- Navigationsjournal

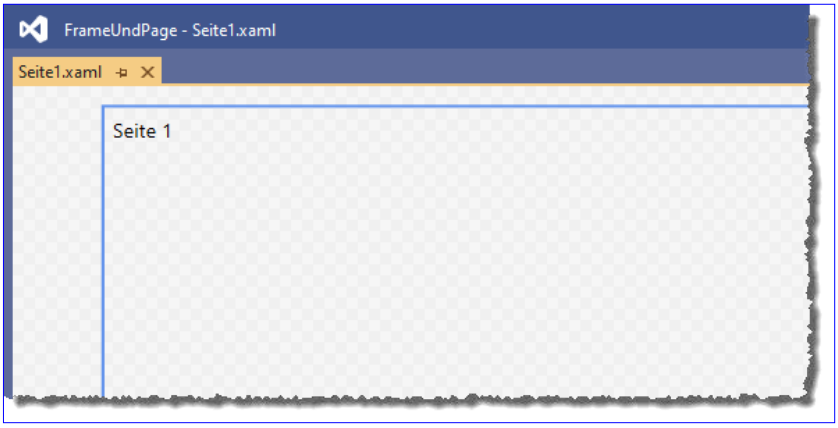


Bild 3: Darstellung eines **Page**-Elements im Entwurf

Page-Element mit Content anzeigen

Die erste Möglichkeit der Anzeige eines **Page**-Elements ist die **Content**-Eigenschaft des **Frame**-Elements.

Bild 4: Anzeigen eines **Page**-Elements im **Frame**-Element

Wir hinterlegen die folgende Methode für die Schaltfläche **btnSeite1**:

```
Private Sub btnSeite1_Click(sender As Object, e As RoutedEventArgs)
    Dim pgeSeite1 As New Seite1
    fra.Content = pgeSeite1
End Sub
```

Diese erstellt eine neue Instanz des **Page**-Elements **Seite1** und weist diese der Eigenschaft **Content** des Frame-Elements namens **fra** zu. Das Ergebnis sehen Sie in Bild 4.

Auf die gleiche Weise können wir auch die übrigen Seiten anzeigen. Eigentlich reicht diese Technik schon aus, wenn Sie einfach nur über ein Ribbon die verschiedenen Elemente der Benutzeroberfläche einer Anwendung einblenden möchten.

Seiten ausblenden beziehungsweise schließen

Vielleicht möchten Sie einen geöffneten Bereich wieder schließen – beispielsweise, indem Sie eine dafür vorgesehene Schaltfläche mit dem Text **Schließen** oder **OK** betätigen.

Eine Möglichkeit ist das Einstellen der **Content**-Eigenschaft des **Frame**-Elements auf den Wert **Nothing**. Damit wird das **Frame**-Element geleert. Fügen wir dem Hauptfenster eine Schaltfläche namens **btnLeeren** hinzu, können wir für ihr **Click**-Ereignis die folgende Methode hinterlegen:

```
Private Sub btnLeeren_Click(sender As Object, e As RoutedEventArgs)
    fra.Content = Nothing
End Sub
```

Wenn wir nun eines der **Page**-Elemente geladen haben und diese Schaltfläche betätigen, wird das **Frame**-Element geleert.

Wenn Sie das **Page**-Element jedoch vom geöffneten **Page**-Element aus schließen wollen, beispielsweise per Schaltfläche, dann haben Sie ebenfalls mehrere Möglichkeiten – zum Beispiel die folgenden:

- Durch Übergabe eines Verweises auf das **Frame**-Steuerelement an die aufgerufene Seite und Leeren der Content-Eigenschaft (siehe nächster Abschnitt).
- Durch Verwenden der **NavigationService**-Klasse (siehe weiter unten unter den Methoden der **NavigationService**-Klasse).

Seite schließen per übergebenem Frame-Verweis

Dieses Beispiel schauen wir uns anhand der Schaltfläche **btnSeite2** des Fensters **MainWindow** und dem **Page**-Element **Seite2** an. Für dieses **Page**-Element legen wir eine Variable namens **m_Frame** an, die einen Verweis auf das übergeordnete **Frame**-Element aufnehmen soll. Außerdem erstellen wir eine Konstruktor-Methode, die einen Verweis auf das **Frame**-Element entgegennimmt und diesen in der Variablen **m_Frame** speichert:

```
Class Seite2
    Private m_Frame As Frame

    Public Sub New(fra As Frame)
```

Nicht-generische Auflistungen unter VB.NET

Unter VB6/VBA waren die bekanntesten und sofort einsetzbaren Auflistungsklassen das Array und die Collection. Sie boten einfache Möglichkeiten zum Hinzufügen von Objektverweisen oder Werten, zum Löschen derselben oder auch zum Bearbeiten. Wer mehr brauchte, konnte sich noch des Dictionary-Objekts bedienen, was aber schon das Einbinden eines weiteren Verweises erforderte. Unter .NET gibt es bei Auflistungsklassen einen wichtigen Unterschied: nicht-generische Auflistungsklassen, die nicht stark typisiert sind und generische Auflistungsklassen, die stark typisiert sind. Der vorliegende Artikel betrachtet die wichtigsten nicht-generischen Auflistungsklassen und ihre Eigenschaften.

Hinweis zu den Beispielen

Die Beispiele dieses Artikels haben wir in LINQPad5 erstellt und ausprobiert. Das ist eine schnellere Möglichkeit als ein Visual Studio Projekt zu erstellen, das Sie für jeden Test neu kompilieren und starten müssen. Weitere Infos zu LINQPad5 finden Sie beispielsweise im Artikel **LINQPad: LINQ, C# und VB einfach ausprobieren** (**www.datenbankentwickler.net/100**).

Wir verwenden hier unter **Language** den Wert **VB Program** und rufen unsere Beispielprozeduren jeweils von der Hauptprozedur **Sub Main** auf.

Was sind Auflistungsklassen?

Auflistungsklassen sind Klassen, mit denen Sie Objekte erstellen können, die das Speichern einer Reihe anderer Objekte erlauben. .NET bietet einige Auflistungsklassen für verschiedene Anwendungszwecke an. Sie sind aber nicht einfach nur Sammlungen von Daten, sondern bieten auch die Infrastruktur an, um diese Daten an verschiedenen Stellen auf spezielle Art zu nutzen – zum Beispiel als Datenquelle von Steuerelementen unter WPF.

Die Auflistungsklassen implementieren verschiedene Interfaces, die Eigenschaften, Methoden und Ereignisse für die unterschiedlichen Zwecke vorsehen. Welche Interfaces eine Auflistungsklasse implementiert, können Sie im Objektkatalog von Visual Studio nachsehen (**Strg + Alt**

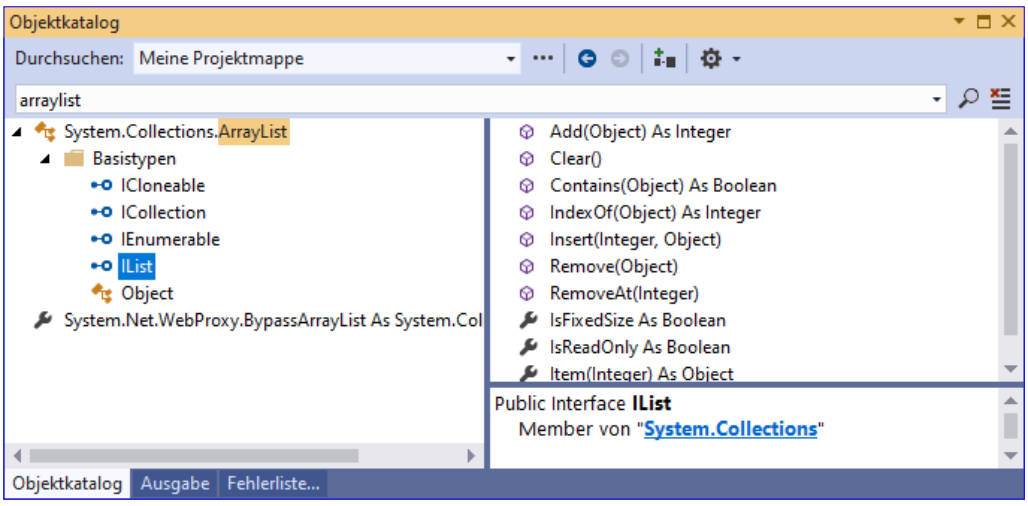


Bild 1: Eigenschaften, Methoden und Ereignisse der **IList**-Schnittstelle

+ J). Bild 1 zeigt die Basistypen beziehungsweise Schnittstellen der Auflistung **ArrayList**. Hier finden Sie auch die Eigenschaften, Methoden und Ereignisse der jeweiligen Auflistungsklasse.

Die **ArrayList**-Auflistungsklasse implementiert beispielsweise die folgenden Interfaces:

- **IEnumerable**: Stellt die Eigenschaft **GetEnumerator** bereit, die allerdings nicht explizit verwendet wird, sondern das Durchlaufen der enthaltenen Elemente in einer **For Each**-Schleife erlaubt
- **ICollection**: Liefert vor allem die Eigenschaft **Count**.
- **IList**: Enthält Methoden, um die Liste zu verwalten – also um Elemente hinzuzufügen, zu entfernen und abzufragen, die Liste zu leeren und weitere Eigenschaften.
- **ICloneable**: Liefert die Methode **Clone**.

Anhand der Schnittstellen, die eine Auflistungsklasse implementiert, können Sie erkennen, welche Möglichkeiten diese bietet. Andere Auflistungsklassen verwenden noch weitere Schnittstellen wie beispielsweise **IDeserializationCallback**, **IDictionary** oder **ISerializable**.

Dadurch, dass die verschiedenen Auflistungsklassen die Schnittstellen implementieren, haben diese einige Gemeinsamkeiten, auf die Sie zählen können – zum Beispiel die Möglichkeit, die Menge der enthaltenen Elemente mit **Count** zu ermitteln oder diese mit **For...Each** zu durchlaufen.

Namespaces mit Auflistungsklassen

Die Auflistungsklassen sind in eigenen Namespaces organisiert. Der Namespace **System.Collections** enthält beispielsweise die Auflistungsklassen **ArrayList, HashTable**, **SortedList** oder **Stack**. Die generischen Auflistungsklassen, über die wir in einem weiteren Artikel namens **Generische Auflistungen unter VB.NET** berichten (**www.datenbankentwickler.net/268**), finden Sie im Namespace **System.Collections.Generic**. Die **ObservableCollection**, die oft als Datenquelle für WPF-Steuerelemente dient, stammt aus dem Namespace **System.Collections.Objectmodel**. Die jeweiligen Schnittstellen sind übrigens auch in den jeweiligen Namespaces zu finden.

Nicht-generische versus generische Auflistungsklassen

Nicht-generische Auflistungsklassen unter VB.Net sind, wie oben bereits angedeutet, nicht »stark typisiert«. Was heißt das überhaupt? Das bedeutet, dass wir nicht explizit den Typ der in der Auflistungsklasse enthaltenen Elemente angeben. Deshalb wird hier der Typ **Object** verwendet. Im Gegensatz dazu geben wir bei generischen Auflistungsklassen den Typ der in der Auflistungsklasse enthaltenen Elemente explizit an, zum Beispiel **String** oder auch komplexere Datentypen. Ein Beispiel, das wir schon oft genutzt haben, sind die Auflistungsklassen mit Entitäten aus einer Datenbank, die wir als Datenquelle für die Steuerelemente unter WPF nutzen – diese haben beispielsweise den Datentyp **Kunde** oder **Produkt**. Nicht generische Auflistungsklassen können Sie hingegen auch Elemente unterschiedlicher Datentypen hinzufügen.

Beispiele für nicht-generische Auflistungsklassen

Es gibt beispielsweise die folgenden nicht-generischen Auflistungsklassen:

- **ArrayList**: eine sortierte Liste von Elementen. Die Sortierung erfolgt nach der Reihenfolge des Hinzufügens der Elemente.
- **Queue**: Die **Queue**-Auflistungsklasse ermöglicht das Verwalten von Einträgen nach dem First-In, First-Out-Prinzip.
- **Stack**: Die **Stack**-Auflistungsklasse arbeitet ähnlich wie Queue, aber andersherum: Sie stellt die zuletzt hinzugefügten Elemente als Erste wieder bereit.
- **HashTable**: Die **HashTable**-Auflistungsklasse speichert Schlüssel-Wert-Paare und bietet umfangreiche Möglichkeiten, diese abzufragen.

Die ArrayList-Auflistung

Wir schauen uns einige Beispiele für den Umgang mit Auflistungsklassen anhand der **ArrayList**-Auflistungsklasse an. Später stellen wir weitere nicht-generische Auflistungsklassen vor und beschreiben deren Besonderheiten.

Die wichtigsten Methoden und Eigenschaften einer **ArrayList** sind die folgenden:

- **Count**: Liefert die Menge der enthaltenen Elemente.
- **CopyTo**: Kopiert den Inhalt der **ArrayList**-Auflistungsklasse in ein Array.
- **Contains**: Erwartet den Wert eines Elements als Parameter und gibt an, ob dieses enthalten ist.
- **ToArray**: Kopiert den Inhalt der **ArrayList**-Auflistungsklasse in ein Array des Typs **Object**.
- **IndexOf**: Liefert den Index des als Parameter angegebenen Elements. Der Index ist immer nullbasiert.
- **GetRange**: Erstellt eine neue **ArrayList**-Auflistungsklasse nur mit den Elementen, die durch den Startindex und die Menge als Parameter angegeben wurden.
- **Sort**: Sortiert die Elemente der ArrayList.
- **Reverse**: Kehrt die Sortierung der Elemente der ArrayList um.
- **Item**: Liest das Element mit dem angegebenen Index aus oder setzt diesen.
- **Clear**: Leert die Auflistungsklasse.

Elemente zu einer nicht-generischen Auflistungsklasse hinzufügen

Die **ArrayList**-Auflistungsklasse ist eine einfache, nicht-generische Auflistungsklasse, daher verwenden wir diese zunächst für die Beispiele. Sie können der **ArrayList** einfach mit der **Add**-Methode neue Elemente hinzufügen, die verschiedenen Typs sein können. Sie können sogar eine neue **ArrayList** als Element der **ArrayList** hinzufügen:

```
Sub ArrayListErstellen
    Dim Gemischt As New ArrayList
    Gemischt.Add(10)
    Gemischt.Add("10")
    Gemischt.Add("André")
    Gemischt.Add(New ArrayList)
End Sub
```

Außerdem sehen Sie, dass wir die Zahl **10** einmal als Zahl und einmal als Zeichenkette hinzugefügt haben. Das weisen die folgenden beiden Anweisungen nach, die Sie der Prozedur hinzufügen:

```
Debug.Print(Gemischt.Item(0).GetType.ToString)
Debug.Print(Gemischt.Item(1).GetType.ToString)
```

Das Ergebnis lautet:

```
System.Int32
System.String
```

Dass Sie nicht wissen, welcher Objekttyp Sie in einem Element einer nicht-generischen Auflistungsklasse erwartet, ist übrigens einer der Gründe, warum Sie eine generische Auflistungsklasse verwenden sollten – außer, Sie benötigen explizit eine nicht-generische Liste.

Sie können auch die Elemente eines Array zu einer **ArrayList**-Auflistungsklasse hinzufügen. Im folgenden Beispiel fügen wir zunächst zwei Elemente wie zuvor per **Add** hinzu. Dann erstellen wir ein Array namens **arr** mit zwei weiteren Elementen. Die Elemente aus diesem Array fügen wir dann mit der **AddRange**-Methode hinzu:

```
Dim Produkte As ArrayList
Dim arr() As String
Produkte = New ArrayList
Produkte.Add("Datenbankentwickler")
Produkte.Add("Access im Unternehmen")
arr = {"Access [basics]", "Access und Formulare"}
Produkte.AddRange(arr)
```

Elemente einer Auflistungsklasse durchlaufen

Die einfachste Methode, die Elemente einer Auflistungsklasse zu durchlaufen, ist die **For Each**-Schleife. Sie verwenden darin eine Variable, welche das jeweilige Element aufnehmen soll und die zu durchlaufende Auflistungsklasse. Die Elemente der soeben angelegten Auflistungsklasse können Sie wie folgt durchlaufen und beispielsweise im Ausgabe-Bereich ausgeben:

```
Dim Produkt As String
For Each Produkt In Produkte
    Debug.Print(Produkt)
Next Produkt
```

Auf Elemente einer Auflistungsklasse per Index zugreifen

Weiter oben haben wir gleich gezeigt, wie Sie über den Index auf die Elemente der Auflistungsklassen zugreifen können. Sie verwenden dazu die **Item**-Eigenschaft und geben den Index des gesuchten Elements an, der immer 0-basiert ist:

```
Debug.Print (Produkte(0))
```

Damit können wir auf einzelne Elemente innerhalb einer weiteren Schleife zugreifen, nämlich **For...Next**:

```
Dim i As Integer
For i = 0 To Produkte.Count - 1
    Debug.Print(Produkte(i))
Next i
```

Elemente an bestimmten Stellen einfügen

Mit der **Add**- oder der **AddRange**-Methode fügen wir Elemente immer hinten an die Liste an. Sie können aber auch Elemente mitten in der **ArrayList**-Auflistungsklasse anfügen. Dazu nutzen Sie die **Insert**- oder die **InsertRange**-Methode. Im folgenden Beispiel fügen wir ein Element mit **Add** an, dann eines mit **Insert** vor diesem Element an der mit dem ersten Parameter angegebenen Position **0**.

Die Elemente des Arrays **arr** fügen wir dann zwischen diese beiden Elemente an Position **1** ein:

```
Dim Produkte As ArrayList
Dim arr() As String
Produkte = New ArrayList
Produkte.Add("Datenbankentwickler")
Produkte.Insert(0, "Access im Unternehmen")
arr = {"Access [basics]", "Access und Formulare"}
Produkte.InsertRange(1, arr)
Produkte.Dump
```

Generische Auflistungen unter VB.NET

Der Unterschied zwischen generischen Auflistungen und nicht-generischen Auflistungen ist, dass Sie für die Elemente einer generischen Auflistung einen konkreten Datentyp vorgeben – das nennt man auch stark typisiert. Bei nicht-generischen Auflistungen nutzen Sie den Objekttyp Object und können somit Werte verschiedener Typen hinzufügen. Dieser Artikel stellt die Grundlagen zu generischen Auflistungsklassen vor und erläutert ihre Funktionsweise.

Hinweis zu den Beispielen

Die Beispiele dieses Artikels haben wir in LINQPad5 erstellt und ausprobiert. Das ist eine schnellere Möglichkeit als ein Visual Studio Projekt zu erstellen, das Sie für jeden Test neu kompilieren und starten müssen. Weitere Infos zu LINQPad5 finden Sie beispielsweise im Artikel **LINQPad: LINQ, C# und VB einfach ausprobieren** (**www.datenbankentwickler.net/100**). Wir verwenden hier unter **Language** den Wert **VB Program** und rufen unsere Beispielprozeduren jeweils von der Hauptprozedur **Sub Main** auf.

Generische Auflistungen

Eine generische Auflistung erkennen Sie primär daran, dass Sie diese mit der Angabe des Datentyps für die enthaltenen Werte deklarieren und initialisieren. Dazu verwenden wir das Schlüsselwort **Of** zusammen mit dem gewünschten Datentyp. Die gängigen generischen Auflistungen werden in den Namespaces **System.Collections.Generic** und **System.Collections.ObjectModel** aufgeführt.

Zu den nicht-generischen Auflistungen gibt es in den meisten Fällen ein Pendant unter den generischen Auflistungen. Diese sind leider nicht entsprechend benannt. So ist das generische Gegenstück zu der nicht-generischen Auflistungsklasse **HashTable** die **Dictionary**-Klasse und **List** ist das Pendant zu **ArrayList**.

Es gibt aber auch generische Auflistungsklassen ohne nicht-generische Entsprechung.

Was sind generische Auflistungsklassen?

Generische Auflistungen sind auch Klassen, allerdings solche, die das Speichern einer Reihe anderer Objekte erlauben. .NET bietet einige Auflistungsklassen für verschiedene Anwendungszwecke an. Sie sind aber nicht einfach nur Sammlungen von Daten, sondern bieten auch die Infrastruktur an, um diese Daten an verschiedenen Stellen auf spezielle Art zu nutzen – zum Beispiel als Datenquelle von Steuerelementen unter WPF. Die Auflistungsklassen implementieren verschiedene Interfaces, die Eigenschaften, Methoden und Ereignisse für

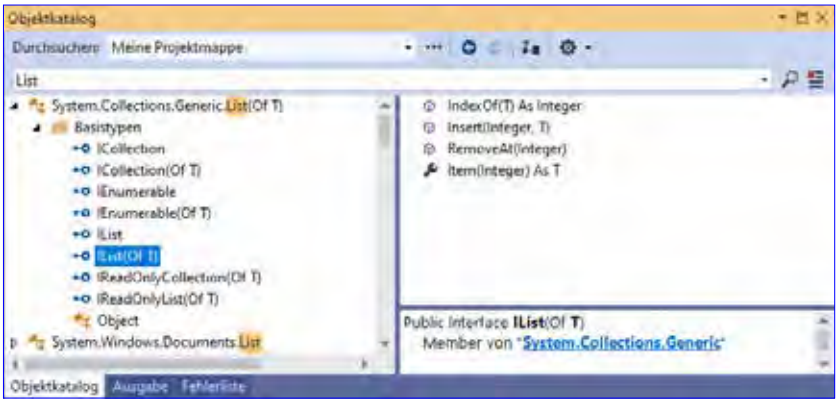


Bild 1: Die **List(Of T)**-Auflistungsklasse im Objektkatalog

die unterschiedlichen Zwecke vorsehen. Welche Interfaces eine Collection implementiert, können Sie im Objektkatalog von Visual Studio nachsehen (**Strg + Alt + J**).

Bild 1 zeigt die Basistypen beziehungsweise Schnittstellen der Auflistungsklasse **List(Of T)**, wobei **T** dem Datentyp der Elemente der Auflistung entspricht. Im Objektkatalog finden Sie auch die Eigenschaften, Methoden und Ereignisse der jeweiligen Auflistungen.

Die **List(Of T)**-Auflistungsklasse implementiert beispielsweise die folgenden Interfaces:

- **IEnumerable**: Stellt die Eigenschaft **GetEnumerator** bereit, die allerdings nicht explizit verwendet wird, sondern das Durchlaufen der enthaltenen Elemente in einer **For Each**-Schleife erlaubt
- **ICollection**: Liefert vor allem die Eigenschaft **Count**.
- **IList**: Enthält Methoden, um die Liste zu verwalten – also um Elemente hinzuzufügen, zu entfernen und abzufragen, die Liste zu leeren und weitere Eigenschaften.

Beispiele für generische Auflistungsklassen

In diesem Artikel stellen wir verschiedene generische Auflistungsklassen vor:

- **List(Of T)**: Entspricht der nicht-generischen Auflistungsklasse **ArrayList**.
- **Dictionary(Of TKey, TValue)**: Entspricht der nicht-generischen Auflistungsklasse **HashTable**.

Die List(Of T)-Auflistungsklasse

Die **List(Of T)**-Auflistungsklasse ist das generische Pendant zur **ArrayList**-Auflistungsklasse, das heißt, dass für Ihre Elemente exakt festgelegt wird, welchen Typ diese aufweisen. Das nennt sich auch stark typisiert.

In den folgenden Beispielen wollen wir mit einer kleinen Beispielklasse namens Produkt arbeiten, die wir wie folgt deklarieren:

```
Public Class Produkt
    Public Produktname As String
    Public Einzelpreis As Decimal
End Class
```

Ein paar Beispielobjekte auf Basis dieser Klasse erstellen wir mit den folgenden Anweisungen:

```
Dim Produkt1 As New Produkt With {.Produktname = "Access im Unternehmen", .Einzelpreis = 124}
Dim Produkt2 As New Produkt With {.Produktname = "Access [Basics]", .Einzelpreis = 69}
Dim Produkt3 As New Produkt With {.Produktname = "Datenbankentwickler", .Einzelpreis = 129}
```

Generische Auflistungsklasse deklarieren und initialisieren

Bei der Deklaration einer generischen Auflistungsklasse geben Sie als Parameter das Schlüsselwort **Of** und den Typ des enthaltenen Objekttyps an. Für unser Beispiel verwenden wir eine Auflistung namens **Produkte** mit dem Typ **Produkt**. Die Deklarationszeile sieht so aus:

```
Dim Produkte As List(Of Produkt)
```

Wenn Sie die Initialisierung separat gestalten wollen, gelingt dies so:

```
Produkte = New List(Of Produkt)
```

Sie können natürlich auch beides in einer Zeile unterbringen:

```
Dim Produkte As New List(Of Produkt)
```

Elemente zu einer generischen Auflistungsklasse hinzufügen

Die oben erstellten Objekte fügen wir mit der **Add**-Methode zur Auflistungsklasse **Produkte** hinzu:

```
With Produkte
    .Add(Produkt1)
    .Add(Produkt2)
    .Add(Produkt3)
End With
```

Generische Auflistungsklasse schnell füllen per Collection Initializer

Wenn Sie die Liste nicht etwa per Entity Framework aus einer **DbSet**-Auflistung füllen wollen, können Sie den sogenannten **Collection Initializer** nutzen. Dabei geben Sie hinter der eigentlichen Initialisierung das **From**-Schlüsselwort und dann in geschweiften Klammern eine Liste der einzufügenden Elemente an:

```
Dim Produkt1 As New Produkt With {.Produktname = "Access im Unternehmen", .Einzelpreis = 124}
Dim Produkt2 As New Produkt With {.Produktname = "Access [Basics]", .Einzelpreis = 69}
Dim Produkt3 As New Produkt With {.Produktname = "Datenbankentwickler", .Einzelpreis = 129}
Dim Produkte As New List(Of Produkt) From {Produkt1, Produkt2, Produkt3}
Debug.Print("Anzahl Produkte: " + Produkte.Count.ToString)
```

Elemente einer generischen Auflistungsklasse mit For Each durchlaufen

Die einfachste Methode, die Elemente einer generischen Auflistungsklasse zu durchlaufen, ist die **For Each**-Schleife. Sie verwenden darin eine Variable, welche das jeweilige Element aufnehmen soll und die zu durchlaufende Auflistungsklasse. Die Elemente der soeben angelegten Auflistungsklasse können Sie wie folgt durchlaufen und beispielsweise im Ausgabe-Bereich ausgeben – ausgehend von einer wie oben beschrieben erstellten Auflistung namens **Produkte**:

```
For Each Produkt In Produkte
    Debug.Print(Produkt.Produktname + " " + Produkt.Einzelpreis.ToString)
Next Produkt
```

Das Ergebnis lautet:

```
Access im Unternehmen 124
Access [Basics] 69
Datenbankentwickler 129
```

Auf Elemente einer generischen Auflistungsklasse per Index zugreifen

Die Elemente einer **List(Of T)**-Auflistungsklasse können Sie genau wie bei der **ArrayList**-Auflistungsklasse auch über die **Item**-Eigenschaft referenzieren. Dazu können Sie beispielsweise den Index als Parameter verwenden. Wichtig ist, dass Sie damit ein Objekt des jeweiligen Typs, hier **Produkt**, zurückbekommen:

```
Dim Produkt4 As Produkt
Produkt4 = Produkte.Item(0)
Debug.Print(Produkt4.Produktname)
```

Da **Item** die Standardeigenschaft ist, können Sie sie auch weglassen und direkt den Index angeben:

```
Produkt4 = Produkte(0)
```

Sie können auch direkt über das **Item**-Element auf die Eigenschaften zugreifen:

```
Dim strProduktname As String
strProduktname = Produkte(0).Produktname
Debug.Print(strProduktname)
```

Elemente einer generischen Auflistungsklasse mit For...Next durchlaufen

Damit können wir sowohl auf einzelne Elemente innerhalb einer Schleife zugreifen, nämlich **For...Next**:

```
Dim i As Integer
For i = 0 To Produkte.Count - 1
    Debug.Print(Produkte(i).Produktname)
Next i
```

Elemente an bestimmter Stelle einfügen

Mit der **Add**-Methode fügen wir Elemente immer hinten an die Liste an. Sie können aber auch Elemente mitten in der **ArrayList**-Auflistungsklasse anfügen. Dazu nutzen Sie die **Insert**- oder die **InsertRange**-Methode. Im

folgenden Beispiel fügen wir ein Element mit **Add** an, dann eines mit **Insert** vor diesem Element an der mit dem ersten Parameter angegebenen Position **0**.

Die Elemente des Arrays **arr** fügen wir dann zwischen diese beiden Elemente an Position **1** ein:

```
Produkte.Insert(1, Produkt3)
```

Mehrere Elemente an bestimmter Stelle einfügen

Mit der **AddRange**-Methode können Sie auch mehrerer Elemente gleichzeitig hinzufügen. Voraussetzung dafür ist, dass die hinzuzufügenden Elemente sich ebenfalls in einer generischen Auflistung befinden. Im folgenden Beispiel erstellen wir eine Zielauflistung und eine, aus der die Elemente zur Zielauflistung hinzugefügt werden – in diesem Fall auch eine des Typs **List(Of T)** namens **ProdukteZumEinfuegen**. Dieser fügen wir zwei Elemente hinzu. Dann fügen wir mit der **InsertRange**-Methode die Elemente aus **ProdukteZumEinfuegen** zur Auflistung Produkte hinzu – und zwar an der **Index**-Position **0**:

```
Dim Produkte As New List(Of Produkt)
Dim ProdukteZumEinfuegen As New List(Of Produkt)
With Produkte
        .Add(Produkt1)
End With
With ProdukteZumEinfuegen
        .Add(Produkt2)
        .Add(Produkt3)
End With
Produkte.InsertRange(0, ProdukteZumEinfuegen)
```

Elemente aus generischen Auflistungsklassen entfernen

Um einzelne oder mehrere Elemente wieder zu entfernen, können Sie drei Möglichkeiten nutzen:

- **Remove**: Erwartet das zu entfernende Element als Parameter, im Fall eines Strings also beispielsweise **"Datenbankentwickler"**.
- **RemoveAt**: Erwartet den Index des zu entfernenden Elements.
- **RemoveRange**: Erwartet als ersten Parameter den Index des ersten zu entfernenden Elements und als zweiten Parameter die Menge der zu entfernenden Elemente.

Im folgenden Beispiel löschen wir nacheinander alle zuvor hinzugefügten Elemente wieder mit den drei Methoden **Remove**, **RemoveAt** und **RemoveRange**.

RemoveAt entfernt das Element mit dem als Parameter angegebenen Index:

```
Produkte.RemoveAt(0)
```

Remove entfernt ein Element, das mit dem zu entfernenden Produkt übereinstimmt – zu ermitteln beispielsweise über die **Produkte**-Auflistung:

```
Produkte.Remove(Produkte(0))
```

RemoveRange entfernt vom als ersten Parameter angegebenen Index aus die mit dem zweiten Parameter angegebene Menge aus der Auflistung:

```
Produkte.RemoveRange(0,2)
```

Gegenüber der **ArrayList** liefert **List(Of T)** noch eine weitere Methode zum Entfernen, nämlich **RemoveAll**. Mit dieser können Sie etwa wie folgt Elemente nach bestimmten Kriterien entfernen:

```
Produkte.RemoveAll(Function(Produkt) Produkt.Produktname.StartsWith("A"))
```

Generische Auflistungsklassen leeren

Fehlt noch der Befehl, um eine generische Auflistungsklasse zu leeren. Das erledigen wir mit der **Clear**-Methode:

```
... 'Produkt1 bis Produkt3 erstellen
Dim Produkte As New List(Of Produkt)
With Produkte
    .Add(Produkt1)
    .Add(Produkt2)
    .Add(Produkt3)
End With
Produkte.Clear
```

Elemente der generischen Auflistungsklasse durchsuchen

Da die generische Auflistungsklasse **List(Of T)** auch die Schnittstelle **IEnumerable(Of T)** implementiert, können Sie auch die dort bereitgestellten Funktionen zum Abfragen von Daten nutzen. Hier ein kleines Beispiel, mit dem wir ein einzelnes Element ermitteln – das erste, dessen Eigenschaft Produktname mit **D** beginnt. Dazu nutzen wir die Funktion **First** und übergeben einen Lambda-Ausdruck zum Ermitteln dieses Elements:

```
... 'Liste Produkte mit Elementen füllen
Dim Produkt4 As Produkt
Produkt4 = Produkte.First(Function(p) p.Produktname.StartsWith("D"))
Debug.Print(Produkt4.Produktname)
```

Dies gibt als Ergebnis **Datenbankentwickler** aus.

Auf die gleiche Weise können Sie auch mehrere Elemente ermitteln:

```
... 'Liste Produkte mit Elementen füllen
Dim ProdukteMitA As List(Of Produkt)
ProdukteMitA = Produkte.FindAll(Function(p) p.Produktname.StartsWith("A"))
For Each p As Produkt In ProdukteMitA
    Debug.Print(p.Produktname)
Next p
```

Generische Auflistungsklassen aufsteigend nach einem Feld sortieren

Mit der **Sort**-Methode können Sie eine generische Auflistungsklasse sortieren. Die folgenden Anweisungen sortieren eine wie in den vorherigen Beispielen gefüllte **List(Of T)** und geben diese in der neuen Reihenfolge aus. Im Gegensatz zu **Sort** für eine nicht-generischen Auflistungsklasse benötigen wir hier allerdings noch einen Parameter für die **Sort**-Funktion. Da **T** unter Umständen eine Klasse ist, die mehrere Eigenschaften bereitstellt, müssen wir **Sort** mitteilen, nach welchen Eigenschaften sortiert werden soll. Dies erledigen wir in Form eines Lambda-Ausdrucks. Diesem teilen wir mit, nach welchem Feld wir überhaupt sortieren wollen. In diesem Ausdruck werden mit der **CompareTo**-Funktion jeweils zwei Elemente **x** und **y** verglichen, und zwar die Eigenschaft **Produktname** von **x** mit der Eigenschaft **Produktname** von **y**:

```
Produkte.Sort(Function(x,y) x.Produktname.CompareTo(y.Produktname))
For Each p In Produkte
    Debug.Print(p.Produktname)
Next p
```

Generische Auflistungsklassen absteigend nach einem Feld sortieren

Für eine absteigende Sortierung vertauschen Sie einfach **x** und **y** im **Lambda**-Ausdruck:

```
Produkte.Sort(Function(x,y) y.Produktname.CompareTo(x.Produktname))
For Each p In Produkte
    Debug.Print(p.Produktname)
Next p
```

Die generische Auflistungsklasse Dictionary(Of TKey, TValue)

Das Gegenstück zu der nicht-generischen Auflistungsklasse **HashTable** ist die Auflistungsklasse **Dictionary(Of TKey, TValue)**. Während bei Auflistungsklassen wie **List(Of T)** nur ein Objekt je Element gespeichert werden kann, sind es bei **HashTable** und **Dictionary(Of TKey, TValue)** derer zwei. Um die **Dictionary(Of TKey, TValue)**-Auflistungsklasse typsicher zu machen, müssen Sie natürlich auch sowohl für den Schlüssel als auch für den Wert den Typ angeben, daher folgen hinter dem **Of**-Schlüsselwort gleich zwei Parameter.

Dies müssen nicht gleich benutzerdefinierte Klassen wie oben **Person** sein, sondern Sie können auch als Schlüssel den Datentyp **String** und für den Wert den Datentyp **Integer** definieren:

CSV-Datei in Klassen importieren

Unter Access war es einfach: Dort haben wir eine CSV-Datei oder auch Excel-Tabellen einfach als verknüpfte Tabelle eingebunden und konnten dann direkt auf die Daten zugreifen. Beispielsweise, um diese dann per Anfügeabfrage in die Zieltabelle zu schreiben. Unter Entity Framework ist das so nicht möglich. Natürlich könnte man das Öffnen der Datei von Hand realisieren und die einzelnen Elemente einlesen. Aber es gibt einen anderen, effizienteren Weg, den wir mit einem NuGet-Paket namens CsvHelper beschreiten können. Diese Klasse erlaubt es, die Zeilen einer CSV-Datei automatisch in die Elemente auf Basis einer entsprechenden Klasse einzulesen. Oder Sie können damit auch die Zeilen durchlaufen und die Inhalte auf Basis des Indexes oder der Spaltenüberschrift einlesen und weiterverarbeiten.

CsvHelper hinzufügen

Das erwähnte NuGet-Paket **CsvHelper** fügen wir über den Nuget-Paket-Manager hinzu, den Sie über den Menüeintrag **Projekt|NuGet-Pakete verwalten...** öffnen. Hier wechseln Sie zum Bereich **Durchsuchen** und geben **csvhelper** in das Suchfeld ein. Es erscheint der Eintrag **CsvHelper**, den wir auswählen und mit einem Klick auf die Schaltfläche **Installieren** zum Projekt hinzufügen (siehe Bild 1).

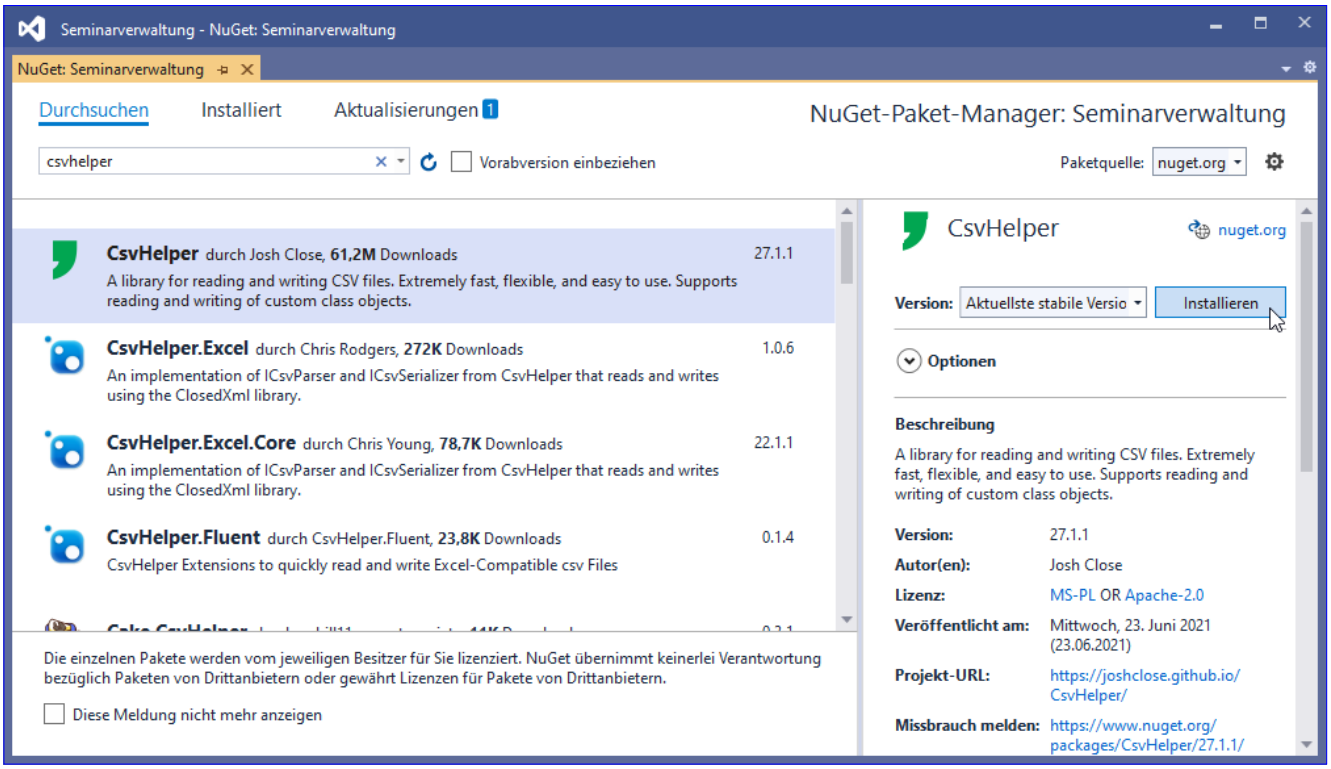


Bild 1: Hinzufügen des **CsvHelper**-Pakets

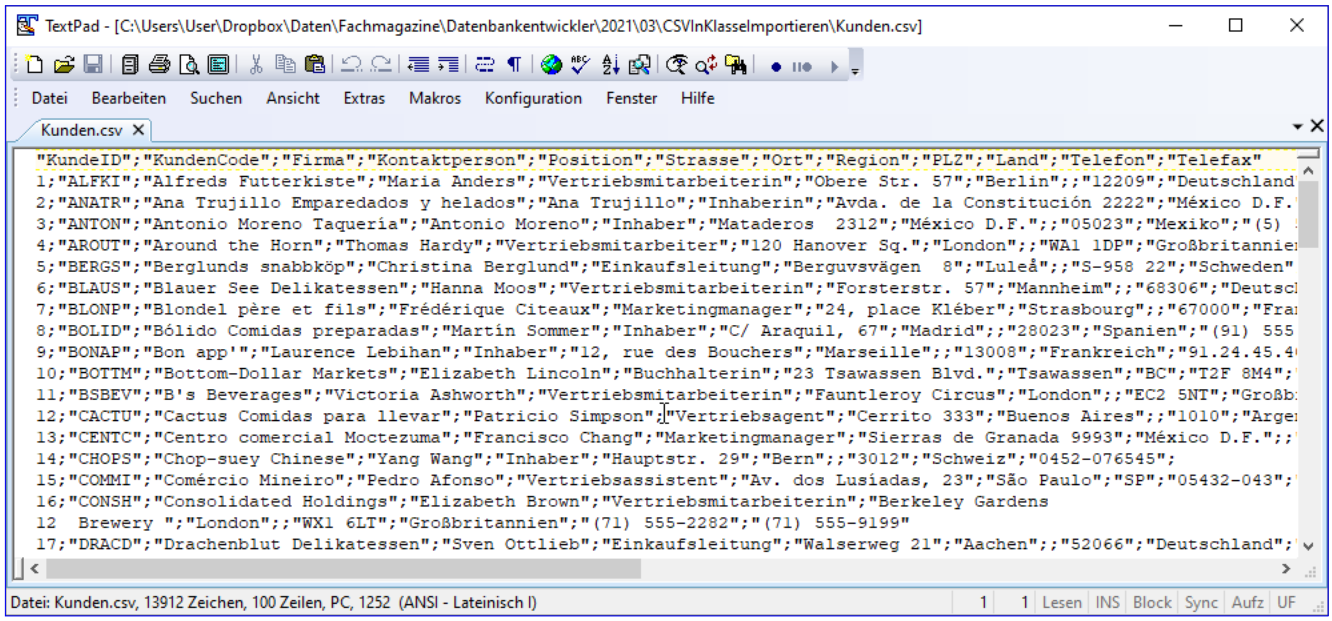


Bild 2: Beispiel-Datei

Beispieldatei für den CSV-Import

Als Beispiel für die folgenden Importtechniken wollen wir eine CSV-Datei verwenden, die wir auf Basis der Tabelle **tblKunden** aus einer Access-Beispieldatenbank exportiert haben (siehe Bild 2).

Zielklasse für die Beispieldaten definieren

Das Ziel ist, die Daten aus einer **.csv**-Datei strukturiert zu erfassen, was in Form des Parsens direkt in Elemente einer Klasse geschehen soll. Dazu benötigen wir natürlich auch eine solche, und zwar eine namens **Kunde**. Diese definieren wir wie folgt:

```
Public Class Kunde
    Public Property KundeID As Long
    Public Property KundenCode As String
    Public Property Firma As String
    Public Property Kontaktperson As String
    Public Property Position As String
    Public Property Strasse As String
    Public Property Ort As String
    Public Property Region As String
    Public Property PLZ As String
    Public Property Land As String
    Public Property Telefon As String
    Public Property Telefax As String
End Class
```

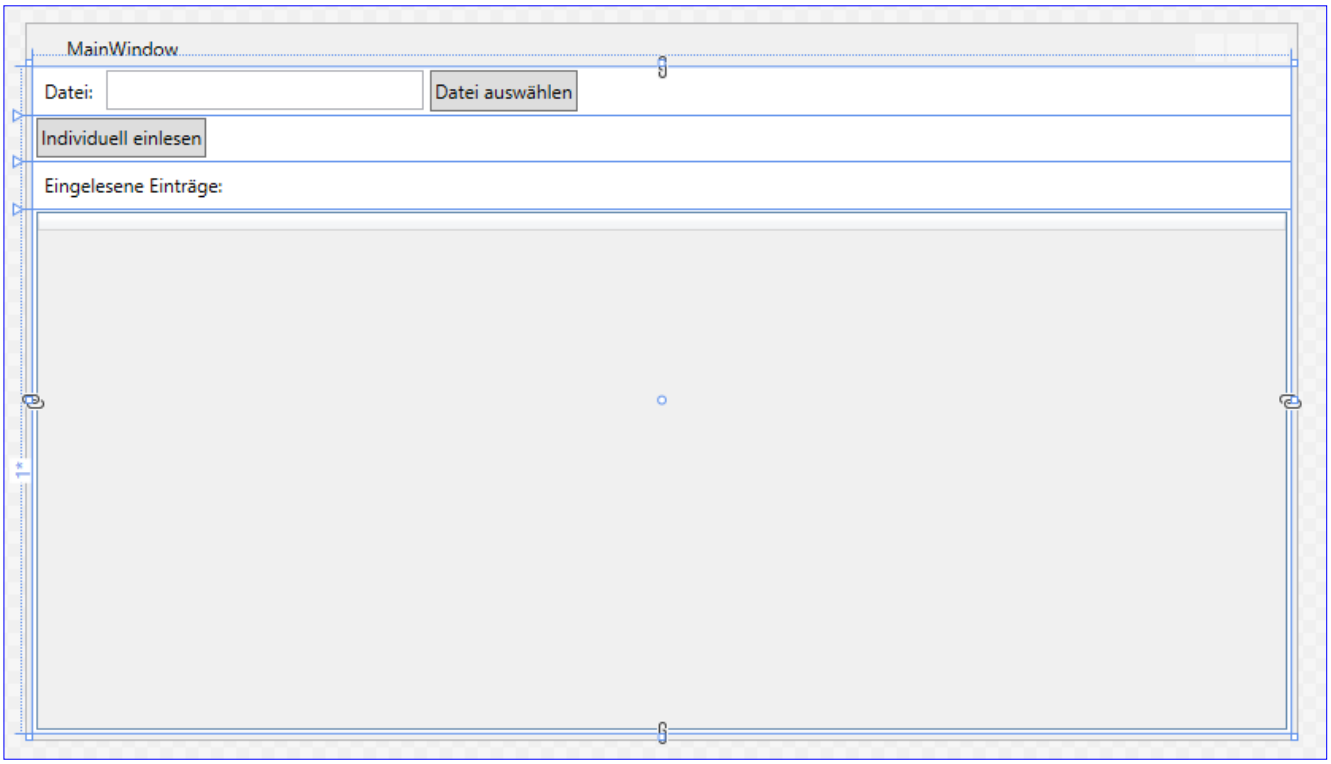


Bild 3: Benutzeroberfläche für die Beispiele

Benutzeroberfläche für die Beispiele

Die Beispiele wollen wir in der Benutzeroberfläche der Anwendung darstellen. Dazu fügen wir einige Elemente hinzu:

- Textfeld **txtImportFile** zur Eingabe/Anzeige der einzulesenden Datei
- Schaltfläche **btnChooseFile** zum Öffnen eines Dateiauswahl-Dialogs
- Schaltfläche **btnImportNoAssignment** zum Starten des ersten Beispiels für den Einlesevorgang
- DataGrid **dgrKunden** zur Anzeige der eingelesenen Daten

Die Steuerelemente ordnen wir wie in Bild 3 an.

Die Elemente definieren wir per XAML-Code wie folgt:

```
<Grid>
   <Grid.RowDefinitions>
      <RowDefinition Height="Auto"></RowDefinition>
      <RowDefinition Height="Auto"></RowDefinition>
```

```
      <RowDefinition Height="*"></RowDefinition>
      <RowDefinition Height="Auto"></RowDefinition>
   </Grid.RowDefinitions>
   <GroupBox Header="Beispiele Artikel CSV-Datei in Klassen importieren" FontWeight="Bold" Grid.Row="0">
      <ContentControl FontWeight="Normal">
         <StackPanel Orientation="Vertical">
            <Grid>
               <Grid.ColumnDefinitions>
                  <ColumnDefinition Width="Auto"></ColumnDefinition>
                  <ColumnDefinition Width="*"></ColumnDefinition>
                  <ColumnDefinition Width="Auto"></ColumnDefinition>
               </Grid.ColumnDefinitions>
               <Label Grid.Column="0">Datei:</Label>
               <TextBox x:Name="txtImportfile" Grid.Column="1" HorizontalAlignment="Stretch"></TextBox>
               <Button x:Name="btnChooseFile" Grid.Column="2"
                  Click="btnChooseFile_Click">Datei auswählen</Button>
            </Grid>
            <StackPanel Orientation="Horizontal">
               <Button x:Name="btnImportNoAssignment"
                  Click="btnImportNoAssignment_Click">Ohne Zuordnung einlesen</Button>
            </StackPanel>
         </StackPanel>
      </ContentControl>
   </GroupBox>
   <Label Grid.Row="1">Eingelesene Einträge:</Label>
   <DataGrid x:Name="dgrKunden" Grid.Row="2"></DataGrid>
</Grid>
```

CSV-Datei auswählen

Um die Datei auszuwählen, kann der Benutzer mit der Schaltfläche **btnChooseFile** einen Dateiauswahldialog öffnen. Für den Code benötigen wir die folgende **Imports Namespace**-Anweisung:

```
Imports Microsoft.Win32
```

Die Schaltfläche löst die folgende Methode aus. Hier stellen wir das Verzeichnis auf das aktuelle Anwendungsverzeichnis ein (also das Verzeichnis, in dem sich die **.exe**-Datei befindet). Außerdem legen wir die Standarddateiendung auf **.csv** fest. Danach öffnen wir den Dialog und lesen das Ergebnis in das Textfeld **txtImportfile** ein:

```
Private Sub btnChooseFile_Click(sender As Object, e As RoutedEventArgs)
    Dim objOpenFileDialog As New OpenFileDialog
    objOpenFileDialog.InitialDirectory = AppDomain.CurrentDomain.BaseDirectory
```

```
    objOpenFileDialog.DefaultExt = ".csv"
    If (objOpenFileDialog.ShowDialog = True) Then
        txtImportfile.Text = objOpenFileDialog.FileName
    End If
End Sub
```

Einlesen mit Spaltenüberschriften, aber ohne Mapping

Die einfachste Art, die Daten aus der CSV-Datei in eine Auflistung mit Elementen des Typs **Kunde** einzulesen, ist die hinter der Schaltfläche **btnImportNoAssignment**.

Für die Anweisungen in dieser Methode benötigen wir einige Namespaces, die wir wie folgt deklarieren:

```
Imports System.Globalization
Imports System.IO
Imports System.Text
Imports CsvHelper
```

In der Methode definieren wir je ein Element der Typen **StreamReader** und **CsvReader**. Dann schreibt die Prozedur den Inhalt des Textfeldes **txtImportfile** in die Variable **strImportFile**:

```
Private Sub btnImportNoAssignment_Click(sender As Object, e As RoutedEventArgs)
    Dim strImportFile As String
    Dim objReader As StreamReader
    Dim objCsvReader As CsvReader
    Dim objRecords As Object
    Dim objKunde As Kunde
    Dim objKunden As New List(Of Kunde)
    strImportFile = txtImportfile.Text
```

Anschließend öffnen wir die CSV-Datei als neues **StreamReader**-Objekt:

```
    objReader = New StreamReader(strImportFile, Encoding.UTF8)
```

Auf dieses greift dann das neue **CsvReader**-Objekt zu und liest den Inhalt mit der **GetRecords**-Methode für den Datentyp **Kunde** in die Variable **objRecords** ein:

```
    objCsvReader = New CsvReader(objReader, CultureInfo.CurrentCulture)
    objRecords = objCsvReader.GetRecords(Of Kunde)()
```

Die in dieser Liste enthaltenen **Kunde**-Objekte schreiben wir dann in eine **List(Of Kunde)**-Auflistung und zeigen diese im **DataGrid**-Element an:

EDM-Daten in CSV exportieren

Im Artikel »CSV-Datei in Klassen importieren« haben wir bereits gezeigt, wie Sie die Daten aus CSV-Dateien in .NET-Anwendungen importieren und diese auf die enthaltenen Klassen aufteilen können. Dazu haben wir das NuGet-Paket CsvHelper verwendet. Im vorliegenden Artikel schauen wir uns den umgekehrten Weg an: Wie können wir die Daten aus den Objekten eines Entity Data Models in eine CSV-Datei exportieren? Dazu setzen wir direkt an der Beispieldatenbank für den oben genannten Artikel an und speichern die darauf importierten Daten direkt wieder in eine neue CSV-Datei.

CsvHelper hinzufügen

Das erwähnte NuGet-Paket **CsvHelper** fügen wir über den Nuget-Paket-Manager hinzu, den Sie über den Menüeintrag **Projekt|NuGet-Pakete verwalten...** öffnen. Hier wechseln Sie zum Bereich **Durchsuchen** und geben **csvhelper** in das Suchfeld ein. Es erscheint der Eintrag **CsvHelper**, den wir auswählen und mit einem Klick auf die Schaltfläche **Installieren** zum Projekt hinzufügen.

Beispielanwendung

Die Benutzeroberfläche der Beispielanwendung bauen wir auf dem Beispiel aus dem oben genannten Artikel auf. Dort finden wir bereits Steuerelemente, mit denen Sie eine CSV-Datei zum Importieren ihrer Inhalte auswählen

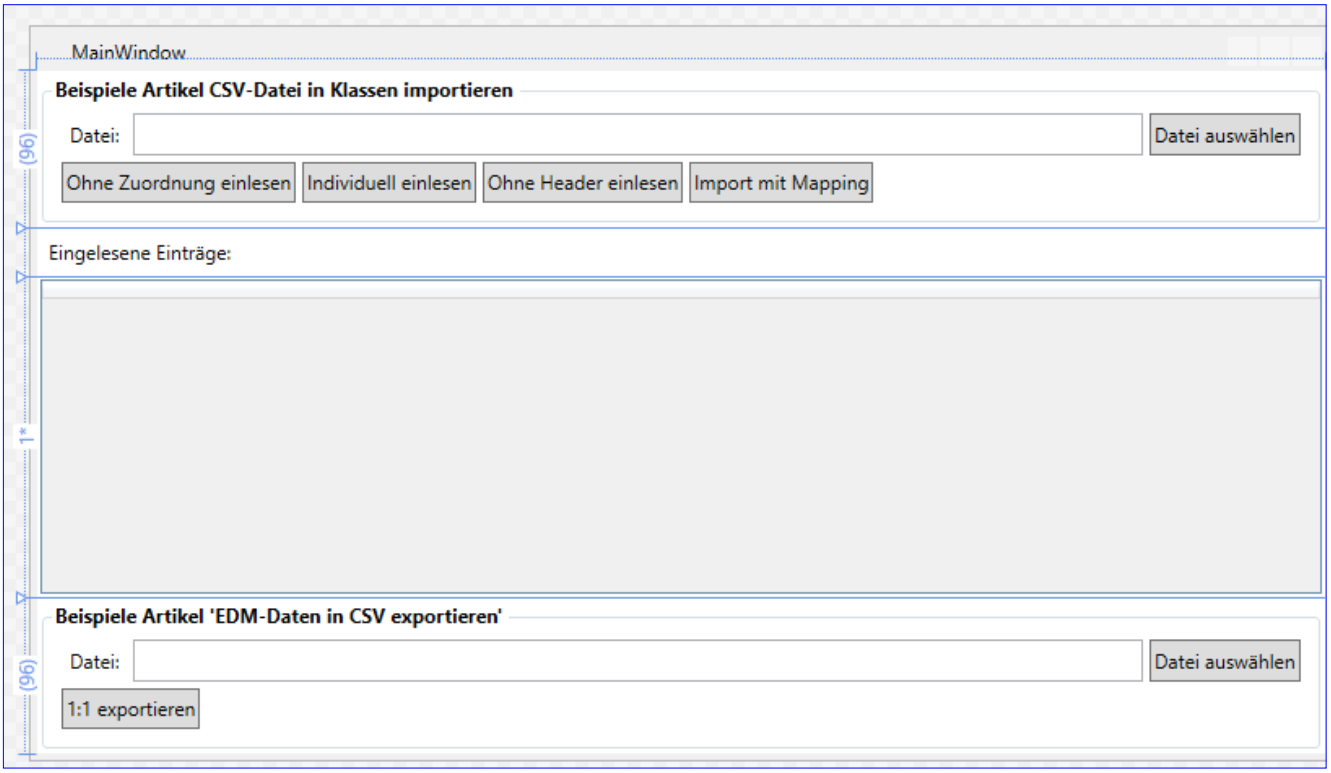


Bild 1: Entwurf des Hauptfensters der Anwendung

können. Die Inhalte werden dann in einem **DataGrid**-Steuerelement angezeigt. Hier fügen wir einen weiteren Bereich an, der die Auswahl eines Dateinamens zum Speichern der Inhalte des **DataGrid**-Elements in einer weiteren CSV-Datei erlaubt. Dieses sieht wie in Bild 1 aus.

Hier ist der XAML-Code der hier enthaltenen Steuerelemente. Wir verwenden das DataGrid **dgrKunden**, um die zu exportierenden Daten zu ermitteln. Das **TextBox**-Steuerelement **txtExportfile** nimmt den Pfad der zu erstellenden Datei auf.

Diesen ermittelt der Benutzer durch einen Klick auf die Schaltfläche **btnChooseSaveFile**. Und der eigentliche Export (die erste Version, es folgen weitere), erfolgt über die Schaltfläche **btnExport11**:

```
    ...
    <Label Grid.Row="1">Eingelesene Einträge:</Label>
    <DataGrid x:Name="dgrKunden" Grid.Row="2"></DataGrid>
    <GroupBox Header="Beispiele Artikel 'EDM-Daten in CSV exportieren'" FontWeight="Bold" Grid.Row="3">
        <ContentControl FontWeight="Normal" HorizontalAlignment="Stretch">
            <StackPanel Orientation="Vertical" HorizontalAlignment="Stretch">
                <Grid>
                    <Grid.ColumnDefinitions>
                        <ColumnDefinition Width="Auto"></ColumnDefinition>
                        <ColumnDefinition Width="*"></ColumnDefinition>
                        <ColumnDefinition Width="Auto"></ColumnDefinition>
                    </Grid.ColumnDefinitions>
                    <Label Grid.Column="0">Datei:</Label>
                    <TextBox x:Name="txtExportfile" Grid.Column="1"></TextBox>
                    <Button x:Name="btnChooseSaveFile" Grid.Column="2"
                        Click="btnChooseSaveFile_Click" HorizontalAlignment="Right">Datei auswählen</Button>
                </Grid>
                <StackPanel Orientation="Horizontal">
                    <Button x:Name="btnExport11" Click="btnExport11_Click">1:1 exportieren</Button>
                </StackPanel>
            </StackPanel>
        </ContentControl>
    </GroupBox>
</Grid>
```

Datei zum Speichern festlegen

Die Schaltfläche **btnChooseSaveFile** löst die folgende Methode aus. Diese erfordert das Vorhanden des folgenden Namespaces:

```
Imports Microsoft.Win32
```

Einfacher Export in eine CSV-Datei

Wir gehen an dieser Stelle davon aus, dass das **DataGrid**-Steuerelement wie in Bild 2 einige Datensätze enthält. In der durch die Schaltfläche **btnExport11** aufgerufenen Methode überprüfen wir diese Voraussetzung. Falls Datensätze enthalten sind, liest sie diese in die **List(Of Kunde)**-Variable **objKunden** ein. Dann erstellt die Prozedur ein neues Objekt des Typs **StreamWriter** und übergibt den Namen der zu erzeugenden Datei als Parameter des Konstruktors.

Anschließend landet der **StreamWriter** als erster Parameter im Konstruktor eines neuen Objekts des Typs **CsvWriter**. Mit **WriteRecords** schreiben wir die Elemente aus **objKunden** in den **CsvWriter** und mit **Flush** sorgen wir dafür, dass der Inhalt aus dem Writer in der Datei landet:

```
Private Sub btnExport11_Click(sender As Object, e As RoutedEventArgs)
    Dim objKunden As List(Of Kunde)
    Dim objWriter As StreamWriter
    Dim objCsvWriter As CsvWriter
    Dim strExportFile As String
    strExportFile = txtExportfile.Text
    If Not dgrKunden.Items.Count = 0 Then
        objKunden = dgrKunden.ItemsSource
        objWriter = New StreamWriter(strExportFile)
        objCsvWriter = New CsvWriter(objWriter, CultureInfo.CurrentCulture)
        With objCsvWriter
            .WriteRecords(objKunden)
```

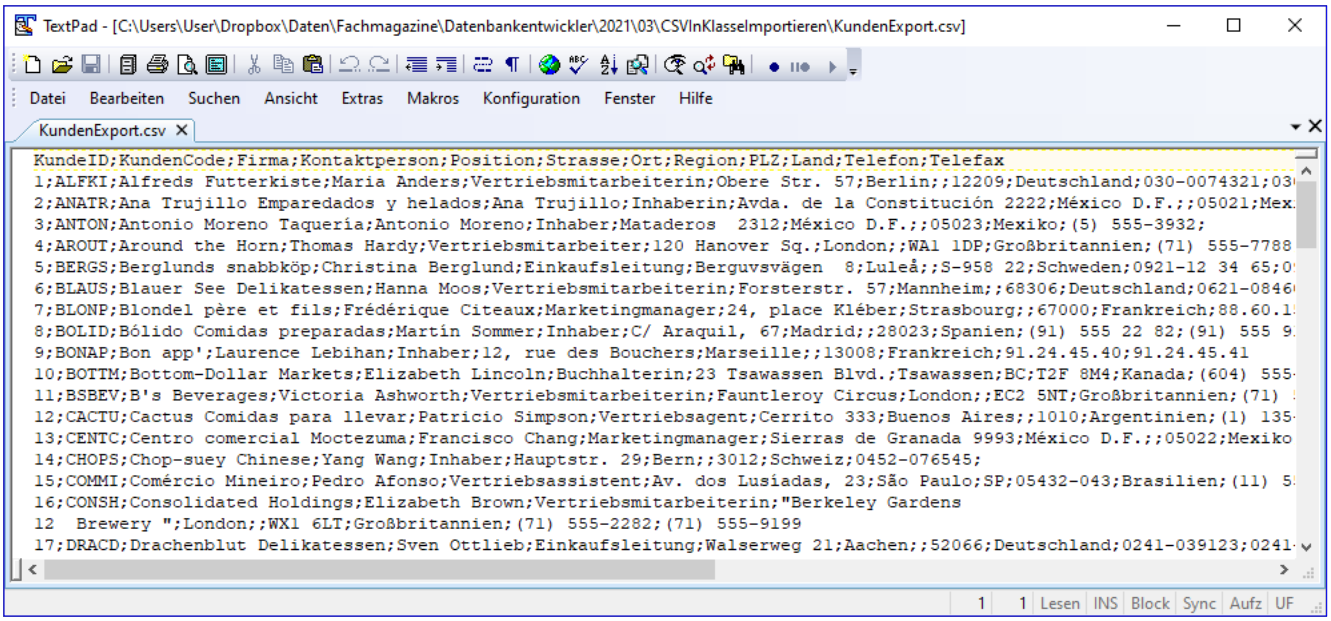


Bild 3: Ergebnis des ersten Exports

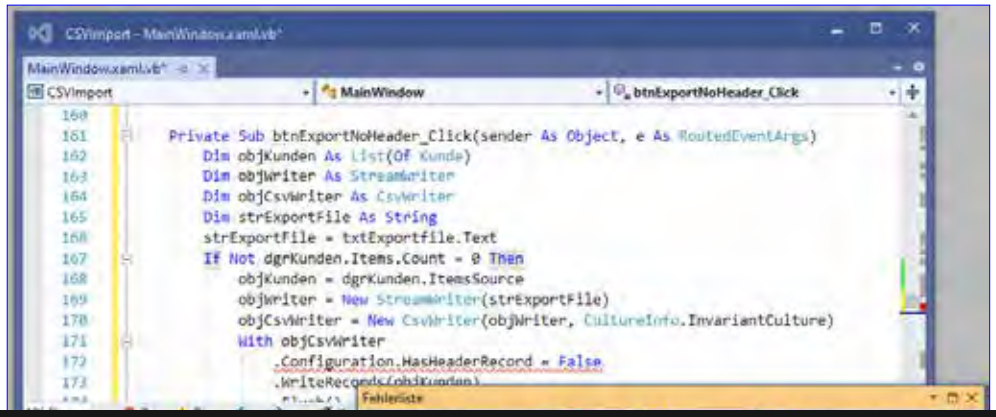
```
            .Flush()
        End With
    Else
        MessageBox.Show("Keine zu exportierenden Daten vorhanden.")
    End If
End Sub
```

Das Ergebnis sehen wir schließlich in Bild 3. Wir sehen hier, dass eine Zeile mit den Eigenschaftsnamen als Spaltenüberschriften exportiert wurde und dass die Elemente durch Semikola voneinander getrennt wurden.

Wir wollen nun ausprobieren, ob man die Spaltenüberschriften ein- und ausblenden kann und wie wir die Bezeichnungen der Spaltenüberschriften selbst anpassen können. All das erledigen wir in den folgenden Beispielen.

Einstellen der Konfiguration für den Export von CSV-Dateien

Wenn wir das Objektmodell des **CsvHelper**-Pakets untersuchen, finden wir für die Klasse **CsvWriter** schnell die Configuration-Klasse mit einigen Eigenschaften wie **HasHeaderRecord**, **Delimiter** et cetera. Wenn wir diese dann allerdings im Code zuweisen wollen, finden wir schnell heraus, dass diese Eigenschaften nur lesbar sind, wir diese aber nicht setzen können (siehe Bild 4).

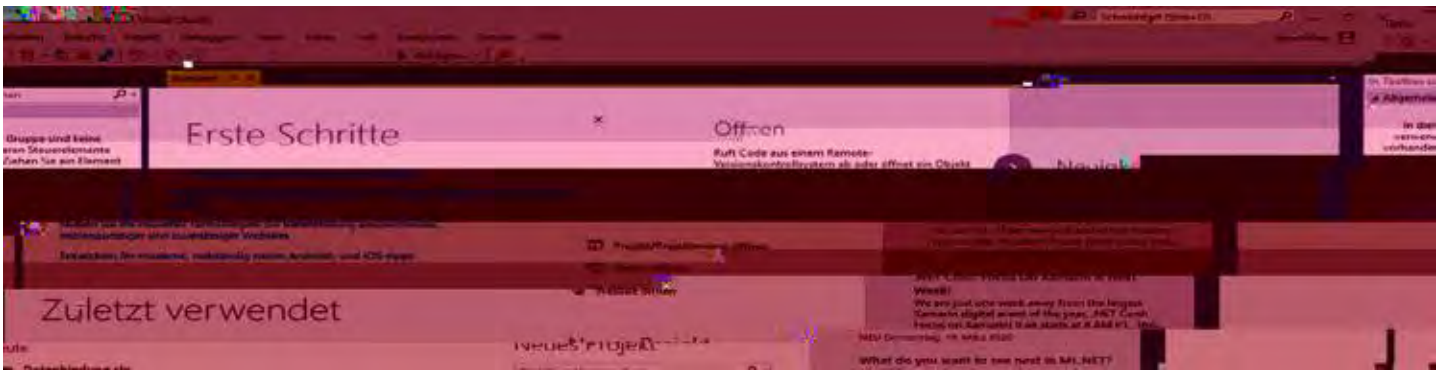


Das ist offensichtlich noch ein Bug, aber zum Glück gibt es einen Workaround. Diesen sehen wir uns in der Methode an, die wir für die Schaltfläche **btnExportNoHea-**

AUSGABE 04/2021

DATENBANK ENTWICKLER

MAGAZIN FÜR DIE DATENBANKENTWICKLUNG MIT VISUAL STUDIO FÜR DESKTOP, WEB UND CO.



TOP-THEMEN:

WPF-BASICS	Von Page zu Page	SEITE 3
	Daten in Listen filtern mit ListCollectionView	SEITE 19
STEUERELEMENTE	Das WPF-Steuerelement CheckBox	SEITE 33
VB-BASICS	Typumwandlung unter VB.NET	SEITE 60

André Minhorst Verlag

Von Page zu Page

Wenn Sie Pages zur Darstellung verschiedener Daten wie Übersichten, Detailansichten oder auch allgemeine Pages nutzen, die in einem Frame-Element angezeigt werden, wollen Sie gegebenenfalls auch einmal Daten von einer Seite zur nächsten schicken. Ein Anwendungsfall wäre die Auswahl eines Artikels in einer Übersichtsseite, der dann in einer Detailseite angezeigt werden soll. Oder das Öffnen einer Übersichtsseite zur Auswahl eines Eintrags, der dann in die aufrufende Seite übernommen werden soll. Dieser Artikel zeigt, wie Sie beim Navigieren im Frame-Element Informationen zwischen Page-Elementen hin- und herschicken können.

Grundgerüst für die Beispiele

Für die Beispiele haben wir im Fenster **MainWindow.xaml** ein Grid mit zwei Zeilen definiert. Es enthält in der ersten Zeile ein **StackPanel**-Element, mit dem wir die einzelnen Beispiele aufrufen können. Die zweite Zeile enthält ein **Frame**-Element namens **fra**, dass die Seiten für die einzelnen Beispiele anzeigen soll:

```
<Grid>
    <Grid.RowDefinitions>
        <RowDefinition Height="Auto"></RowDefinition>
        <RowDefinition Height="*"></RowDefinition>
        <RowDefinition Height="Auto"></RowDefinition>
    </Grid.RowDefinitions>
    <StackPanel Orientation="Horizontal">
        <Button x:Name="btnVonANachB" Click="btnVonANachB_Click">Beispiel Von A nach B per Konstruktor
            starten</Button>
    </StackPanel>
    <Frame x:Name="fra" Grid.Row="1"></Frame>
</Grid>
```

Von A nach B per Konstruktor-Parameter

Wir wollen nun zwei Seiten definieren namens **VonANachB_Konstruktor_SeiteA.xaml** und **VonANachB_Konstruktor_SeiteB.xaml**. Die erste enthält ein Textfeld, in das Sie einen Text ein geben können. Mit einer Schaltfläche wollen wir die zweite Seite aufrufen und dieser den eingegebenen Text übergeben, sodass dieser dort in einem weiteren Textfeld angezeigt werden kann.

Die erste Seite **VonANachB_Konstruktor_SeiteA.xaml** definieren wir wie folgt:

```
<Grid>
    ...
    <Label FontWeight="Bold">Von A nach B per Konstruktor - Seite A</Label>
    <StackPanel Orientation="Horizontal" Grid.Row="1">
```

```
        <Label>Zu übergebender Wert:</Label>
        <TextBox x:Name="txtWert" Width="200"></TextBox>
    </StackPanel>
    <StackPanel Grid.Row="2">
        <Button x:Name="btnVonANachB" HorizontalAlignment="Left" Click="btnVonANachB_Click">
            Wert nach B senden</Button>
    </StackPanel>
</Grid>
```

Die erste Seite sieht in der Entwurfsansicht wie in Bild 1 aus.

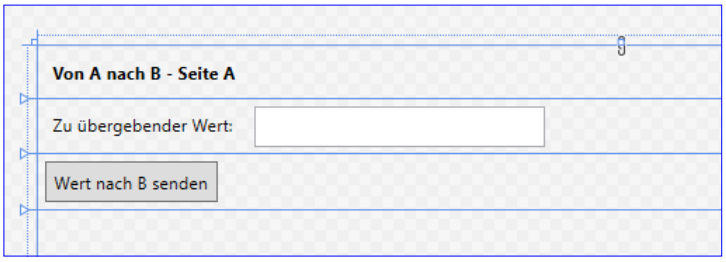


Bild 1: Aufbau der ersten Seite

Die zweite Seite ist noch einfacher aufgebaut – sie enthält lediglich ein **Label**- und ein **TextBox**-Steuerelement:

```
<Grid>
    ...
    <Label FontWeight="Bold">Von A nach B per Konstruktor - Seite B</Label>
    <TextBox x:Name="txtParameter" Grid.Row="1" HorizontalAlignment="Left" Width="200"></TextBox>
</Grid>
```

Erste Seite aufrufen

Um die erste Seite im **Frame**-Element anzuzeigen, betätigen wir die Schaltfläche **btnVonANachB_Konstruktor**. Diese erstellt die Seite **VonANachB_Konstruktor_SeiteA** neu und referenziert diese Instanz mit der Variablen **objVonANachB_Konstruktor_SeiteA**. Damit diese im **Frame**-Element erscheint, rufen wir dessen **Navigate**-Methode mit dem Objektnamen als Parameter auf:

```
Class MainWindow
    Private Sub btnVonANachB_Konstruktor_Click(sender As Object, e As RoutedEventArgs)
        Dim pgeVonANachB_Navigate_SeiteA As New VonANachB_Konstruktor_SeiteA
        fra.Navigate(pgeVonANachB_Navigate_SeiteA)
    End Sub
End Class
```

Konstruktor der zweiten Seite mit Parameter ausstatten

Wir wollen den Wert aus dem Textfeld **txtWert** der ersten Seite zur zweiten Seite schicken. Für die Übergabe wollen wir einen Parameter der Konstruktor-Methode der zweiten Seite nutzen. Die Konstruktormethode, die immer **New** heißt, erstellen Sie am einfachsten, indem Sie mit der rechten Maustaste in die Klasse **VonANach_Konstruktor_B** im Codefenster der Code behind-Klasse **VonANachB_Konstruktor_SeiteB.xaml.vb** klicken und dort den Eintrag **Schnellaktionen und Refactorings...** auswählen.

Nach diesem Befehl erscheint eine Liste der möglichen Schnellaktionen. Hier wählen wir den Eintrag **Konstruktor generieren...** aus (siehe Bild 2).

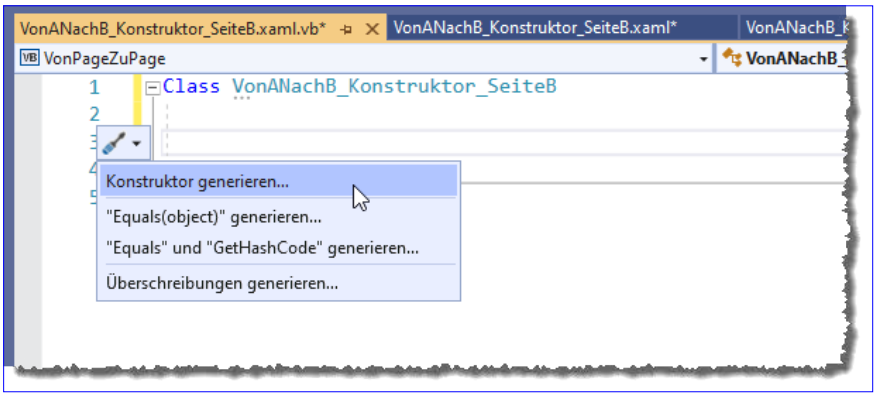


Bild 2: Konstruktor hinzufügen

Danach erscheint noch ein weiterer Dialog namens **Member auswählen**. Hier wählen Sie den Eintrag **_contentLoaded** ab (siehe Bild 3) und betätigen dann die **OK**-Schaltfläche.

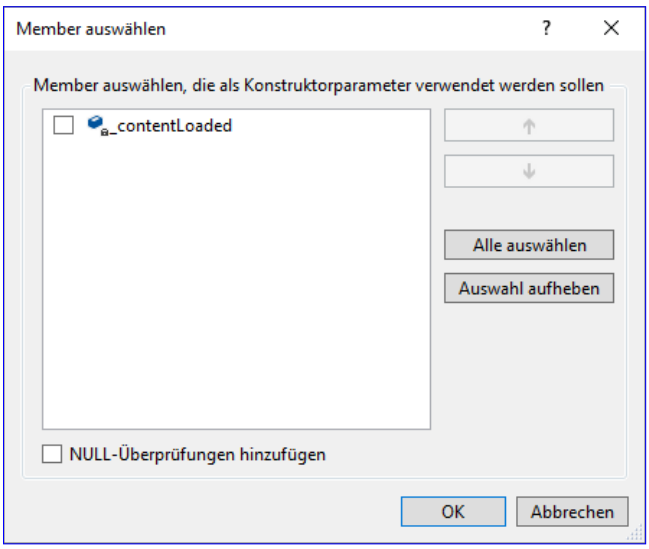


Bild 3: Dialog mit Eigenschaften des Konstruktors

Die Klasse mit der so erstellten Konstruktormethode sieht zunächst wie folgt aus:

```
Class VonANachB_Konstruktor_SeiteB
    Public Sub New()
    End Sub
End Class
```

Diese Methode wollen wir allerdings dazu nutzen, beim Aufruf einen Parameter zu übergeben. Deshalb erweitern wir die erste Zeile um den Parameter **strParameter** mit dem Datentyp **String**. Außerdem fügen wir der Konstruktormethode die Anweisung **InitializeComponent** hinzu, die beim Fehlen einer Konstruktormethode normalerweise implizit ausgeführt wird. Lassen Sie diese weg, wird die XAML-Definition nicht umgesetzt!

Schließlich schreiben wir den Inhalt des Parameters **strParameter** in das Textfeld **txtParameter**:

```
Public Sub New(strParameter As String)
    InitializeComponent()
    txtParameter.Text = strParameter
End Sub
```

Nun fehlt noch die Übergabe des Parameters. Diese erfolgt, wenn der Benutzer die Schaltfläche **btnVonANachB_Konstruktor** in der Klasse **VonANachB_Konstruktor_SeiteA** betätigt.

Die dadurch ausgelöste Methode definieren wir wie folgt:

```
Class VonANachB_Konstruktor_SeiteA
    Private Sub btnVonANachB_Konstruktor_Click(sender As Object, e As RoutedEventArgs)
        Dim pgeVonANachB_Konstruktor_SeiteB As VonANachB_Konstruktor_SeiteB
        Dim strParameter As String
        strParameter = txtWert.Text
        pgeVonANachB_Konstruktor_SeiteB = New VonANachB_Konstruktor_SeiteB(strParameter)
        NavigationService.Navigate(pgeVonANachB_Konstruktor_SeiteB)
    End Sub
End Class
```

Hier deklarieren wir die Zielseite **VonANachB_Konstruktor_SeiteB** mit der Variablen **pgeVonANachB_Konstruktor_SeiteB**. Außerdem deklarieren wir eine Variable namens **strParameter** für den Inhalt des Textfeldes **txtWert**, dass wir direkt in der folgenden Anweisung füllen. Danach erstellen wir eine neue Instanz von **VonANachB_Konstruktor_SeiteB** und übergeben der Konstruktormethode dabei den Wert von **strParameter**. Schließlich navigieren wir mit der **Navigate**-Methode des **NavigationService**-Objekts zu der mit **pgeVonANachB_Konstruktor_SeiteB** referenzierten Seite.

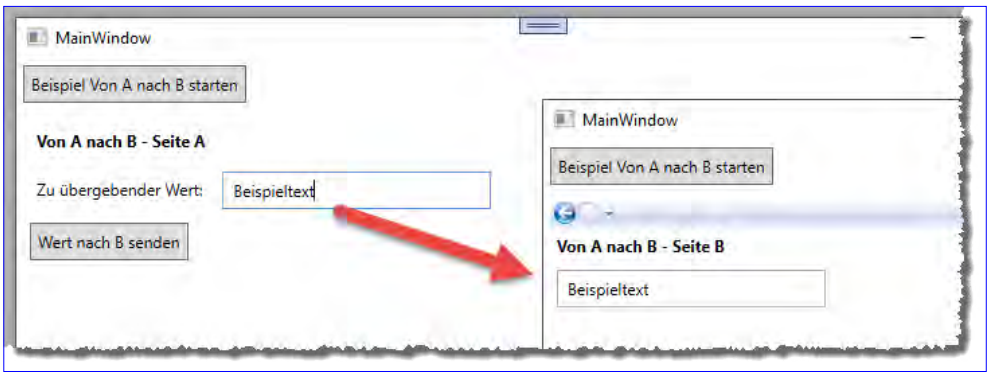


Bild 4: Übergeben eines Textes

Das Ergebnis sehen Sie in Bild 4. Den übergebenen Wert können Sie für beliebige Einsatzzwecke nutzen – beispielsweise für die Übergabe des Primärschlüsselwertes eines in einer Übersicht auf der ersten Seite ausgewählten Eintrags, dessen Detaildaten dann auf der zweiten Seite angezeigt werden sollen.

Von A nach B per Application.Properties

Im zweiten Beispiel nutzen wir keine explizite Übergabe des Wertes von Seite A nach Seite B, sondern wir speichern den zu übergebenden Wert in einer Eigenschaft der aktuellen Anwendung. Der Aufbau ist ähnlich wie im ersten Beispiel, allerdings heißen die beiden Seiten nun **VonANachB_Property_SeiteA** und **VonANachB_Property_SeiteB**. Auf Seite A befindet sich wieder ein Textfeld namens **txtWert**. Außerdem haben wir dort eine Schaltfläche namens **btnVonANachBProperty** angelegt, für die wir die folgende Methode hinterlegt haben:

```
Class VonANachB_Property_SeiteA
    Private Sub btnVonANachB_Property_Click(sender As Object, e As RoutedEventArgs)
        Dim pgeVonANachB_Property_SeiteB As VonANachB_Property_SeiteB
        My.Application.Properties("Parameter") = txtWert.Text
```

Fensterposition beim Öffnen einstellen

Nachdem ich mittlerweile mit zwei Bildschirmen arbeite (einer für die Hauptanwendungen, einer für andere Informationen wie Termine et cetera) und der Bildschirm für die Hauptanwendungen der zweite Bildschirm ist, nervt es mich etwas, dass Anwendungen beim Debuggen immer auf dem sekundären Bildschirm angezeigt werden und ich diese immer erst auf den Hauptbildschirm ziehen muss. Also schauen wir uns in diesem Artikel an, welche Möglichkeiten .NET bietet, festzulegen, auf welchem Bildschirm eine Anwendung geöffnet werden soll.

Ausgangssituation

Die aktuelle Situation ist also, dass ich zwei Bildschirme habe – der linke Bildschirm zeigt unwichtige Anwendungen an, der rechte (oder auch zentrale) Bildschirm dient zum Arbeiten. Wenn ich nun Visual Studio auf dem zentralen Bildschirm geöffnet habe und eine Anwendung zum Debuggen starte, erscheint diese immer auf dem linken, sekundären Bildschirm. Wenn ich die beim Debuggen erstellte **.exe**-Datei direkt aufrufe, wird diese wie gewünscht im Hauptbildschirm angezeigt. Wie kann ich nun steuern, welcher Bildschirm die Anwendung beim Debuggen anzeigt?

Lösung aus der Forms-Bibliothek

Die notwendigen Elemente für den Zugriff auf die Bildschirme finden sich in der Bibliothek **System.Windows.Forms**, die wir zunächst über einen entsprechenden Verweis zu unserem WPF-Projekt hinzufügen. Das erledigen wir über den Dialog, der nach dem Aufruf des Menüeintrags **Projekt|Verweise hinzufügen...** erscheint. Hier sehen Sie im Bereich **Assemblys** nach einer schnellen Suche den Eintrag **System.Windows.Forms**, den Sie per Klick in das Kontrollkästchen zum Projekt hinzufügen (siehe Bild 1). Anschließend können wir per Code auf die Elemente dieser Bibliothek zugreifen.

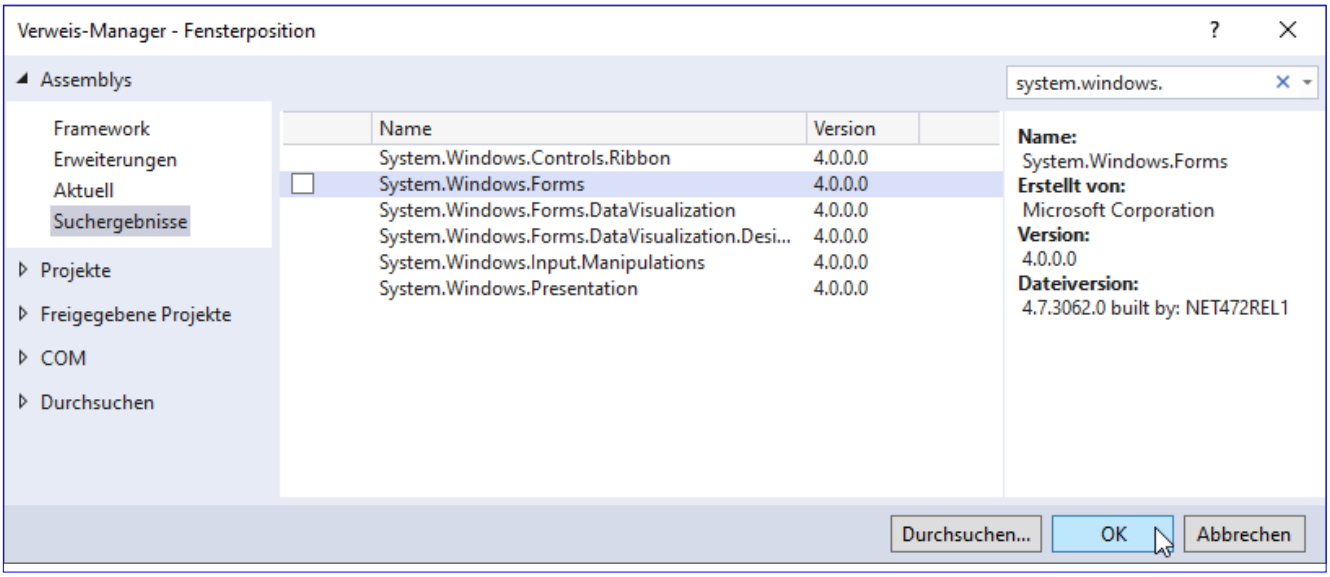


Bild 1: Hinzufügen eines Verweises auf die Bibliothek **System.Windows.Forms**

Die Screen-Klasse

Mit der **Screen**-Klasse eröffnen wir uns die Möglichkeit, auf die Bildschirme des Systems zuzugreifen. Die **Screen**-Klasse liefert uns die folgenden interessanten Auflistungen und Eigenschaften:

- **AllScreens**: Zugriff auf alle **Screen**-Elemente
- **BitsPerPixel**: Anzahl der Bits im Arbeitsspeicher, die einem Pixel von Daten zugeordnet sind
- **Bounds**: Ermittelt die Maße des Bildschirms. Dies liefert ein Objekt des Typs **Rectangle**, das wiederum die vier Eigenschaften **Left**, **Top**, **Width** und **Height** aufweist. Bei mehreren Bildschirmen werden die Werte bezogen auf das Gesamtsystem betrachtet. Wenn der linke Bildschirm beispielsweise eine Breite (**Width**) von 1920 Pixeln hat, dann ist der Wert für die Eigenschaft **Left** des rechten Bildschirms ebenfalls gleich 1920 Pixel.
- **DeviceName**: Liefert den Namen des Bildschirms im System.
- **Primary**: Gibt an, ob der Bildschirm der primäre Bildschirm ist.
- **PrimaryScreen**: Liefert einen Verweis auf den primären Bildschirm.
- **WorkingArea**: Ermittelt die Maße des Arbeitsbereichs des Bildschirms. Dieser unterscheidet sich von den mit Bounds ermittelten Maßen, in dem es nur den Bereich ohne Elemente wie die Taskleiste oder angedockte Fenster und Menüs ermittelt – also den tatsächlich verfügbaren Arbeitsbereich. Die Eigenschaft **WorkingArea** liefert ein Objekt des Typs Rectangle, das wiederum die vier Eigenschaften **Left**, **Top**, **Width** und **Height** aufweist. Auch hier gilt: Bei mehreren Bildschirmen werden die Werte bezogen auf das Gesamtsystem betrachtet. Wenn der linke Bildschirm beispielsweise eine Breite (**Width**) von 1920 Pixeln hat, dann ist der Wert für die Eigenschaft **Left** des rechten Bildschirms ebenfalls gleich 1920 Pixel.

Hier ist nun zu beachten, dass **Screen** selbst bereits ein Element des Typs **Screen** liefert, mit dem Sie einige Eigenschaften des Bildschirms wie **BitsPerPixel**, **Bounds**, **DeviceName**, **Primary** und **WorkingArea** auslesen können. Gleichzeitig können Sie aber über die Eigenschaft **PrimaryScreen** einen Verweis auf das **Screen**-Objekt im System ermitteln, das als primärer Bildschirm angegeben ist.

Für den Zugriff auf die **Screen**-Klasse benötigen wir auch noch die folgende **Imports**-Anweisung:

```
Imports System.Windows.Forms
```

Beispiel: Ausgabe der Bildschirmdaten per Meldungsfenster

Die folgende Methode, die wir in der Konstruktor-Methode des Fensters **MainWindow** der Beispielanwendung aufrufen, definiert eine Variable namens **objScreen** für ein Objekt des Typs **Screen**. In einer **For Each**-Schleife durchlaufen wir alle Elemente der Auflistung **AllScreens** des **Screen**-Objekts. Innerhalb der Schleife setzen wir eine Meldung zusammen, die alle relevanten Informationen über den Bildschirm enthält und zeigen diese dann

in einer Meldung an. Dabei greifen wir auf die einzelnen Werte der Eigenschaften **Bounds** und **WorkingArea** über deren Untereigenschaften **Left**, **Top**, **Width** und **Height** zu. Damit das möglich ist, benötigen Sie noch einen Verweis auf die Bibliothek **System.Drawing**:

```
Public Sub New()
    InitializeComponent()
    Dim objScreen As Screen
    For Each objScreen In Screen.AllScreens
        MessageBox.Show("DeviceName: " & objScreen.DeviceName & vbCrLf _
            & "Primary: " & objScreen.Primary & vbCrLf _
            & "Bounds: " & objScreen.Bounds.Left & ", " & objScreen.Bounds.Top & ", " _
            & objScreen.Bounds.Width & ", " & objScreen.Bounds.Height & vbCrLf _
            & "WorkingArea: " & objScreen.WorkingArea.Left & ", " & objScreen.WorkingArea.Top & ", " _
            & objScreen.WorkingArea.Width & ", " & objScreen.WorkingArea.Height & vbCrLf _
            & "BitsPerPixel: " & objScreen.BitsPerPixel)
    Next
End Sub
```

Das Ergebnis ist je eine Meldung für jeden Bildschirm, der am aktuellen Rechner angeschlossen ist (siehe Bild 2).

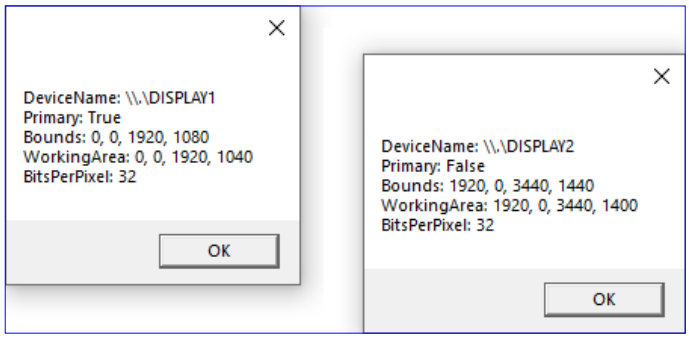


Bild 2: Ausgabe der Daten der beiden Bildschirme des Beispielsystems

Daten der Bildschirme im ListView ausgeben

Weil wir die Daten auch übersichtlich in einem ListView anzeigen wollen, fügen wir dem Fenster MainWindow.xaml die folgende Definition für ein ListView-Steuerelement hinzu:

```
<ListView x:Name="lvwScreens">
    <ListView.View>
        <GridView>
            <GridViewColumn Header="DeviceName" DisplayMemberBinding="{Binding DeviceName}" />
            <GridViewColumn Header="Primary" DisplayMemberBinding="{Binding Primary}" />
            <GridViewColumn Header="Bounds" DisplayMemberBinding="{Binding Bounds}" />
            <GridViewColumn Header="WorkingArea" DisplayMemberBinding="{Binding WorkingArea}" />
            <GridViewColumn Header="PrimaryScreen" DisplayMemberBinding="{Binding PrimaryScreen}" />
            <GridViewColumn Header="BitsPerPixel" DisplayMemberBinding="{Binding BitsPerPixel}" />
        </GridView>
    </ListView.View>
</ListView>
```

Daten in Listen filtern mit ListCollectionView

Daten aus einer SQL Server-Datenbank oder anderen Datenquellen können wir über das Entity Framework als Objekte zugänglich machen und auch als solche in Listen-Steuerelementen anzeigen. Aber wie sieht es mit weiteren Funktionen aus – etwa zum Filtern, Sortieren oder Gruppieren? Dazu benötigen wir einige spezielle Techniken, von denen wir die zum Filtern von Daten nach bestimmten Kriterien im vorliegenden Artikel vorstellen. Dabei nutzen wir Filter für Felder mit verschiedenen Datentypen und in verschiedenen Kombinationen.

Beispieldaten

Als Beispielmaterial wollen wir diesmal nicht auf ein extra anzulegendes Entity Data Model zugreifen, sondern einfach eine Liste von **Kunde**-Elementen erstellen, die ihre Daten aus einer Funktion bezieht statt aus einer Datenbank. Damit wir ein wenig Spielmaterial haben, soll die **Kunde**-Klasse String-, Zahlen-, Datums- und Boolean-Werte enthalten. Die Klasse **Kunde** sieht daher wie folgt aus:

```
Partial Public Class Kunde
    Public Property ID As System.Int32
    Public Property Firma As System.String
    Public Property Vorname As System.String
    Public Property Nachname As System.String
    Public Property Anrede As System.String
    Public Property Strasse As System.String
    Public Property PLZ As System.String
    Public Property Ort As System.String
    Public Property Land As System.String
    Public Property EMail As System.String
    Public Property Geburtsdatum As System.DateTime
    Public Property Alter As System.Int32
    Public Property Newsletter As System.Boolean
End Class
```

Um ein **ListView**-Steuerelement zu füllen, benötigen wir Beispieldaten, die wir mit einer Funktion namens **GetKunden** bereitstellen. Diese fügen wir im Code behind-Modul des Fensters **MainWindow** ein:

```
Private Function GetKunden() As ObservableCollection(Of Kunde)
    Dim Kunden As New ObservableCollection(Of Kunde)
    Kunden.Add(New Kunde() With {.ID = 1, .Firma = "Krahn GbR", .Vorname = "Adi", _
       .Nachname = "Stratmann", .Anrede = "Herr", .Strasse = "Kremser Straße 54", .PLZ = "10589", _
       .Ort = "Berlin", .Land = "Deutschland", .EMail = "adi@stratmann.de", _
       .Geburtsdatum = "20.02.1939", .Alter = 82, .Newsletter = -1})
```

```
    Kunden.Add(New Kunde() With {.ID = 2, .Firma = "Göllner AG", .Vorname = "Heidi", _
        .Nachname = "Eich", .Anrede = "Frau", .Strasse = "Moosstraße 30", .PLZ = "42289", _
        .Ort = "Wuppertal", .Land = "Deutschland", .EMail = "heidi@eich.de", _
        .Geburtsdatum = "30.11.1971", .Alter = 50, .Newsletter = 0})
    Kunden.Add(New Kunde() With {.ID = 3, .Firma = "Peukert GmbH & Co. KG", .Vorname = "Wernfried", _
        .Nachname = "Birk", .Anrede = "Herr", .Strasse = "Wiener Straße 78", .PLZ = "22297", _
        .Ort = "Hamburg", .Land = "Deutschland", .EMail = "wernfried@birk.de", _
        .Geburtsdatum = "23.01.1931", .Alter = 90, .Newsletter = -1})
    ...
    Kunden.Add(New Kunde() With {.ID = 100, .Firma = "Krahn KG", .Vorname = "Rosemaria", _
        .Nachname = "Grill", .Anrede = "Frau", .Strasse = "Kalvarienbergstraße 35", .PLZ = "09131", _
        .Ort = "Chemnitz", .Land = "Deutschland", .EMail = "rosemaria@grill.de", _
        .Geburtsdatum = "02.10.2009", .Alter = 12, .Newsletter = -1})
    Return Kunden
End Function
```

In der Klasse **MainWindow** hinterlegen wir außerdem noch eine private Variable für die Auflistung der Kunden:

```
Private _Kunden As ObservableCollection(Of Kunde)
```

Diese machen wir über die folgende öffentlich deklarierte Property verfügbar:

```
Public Property Kunden As ObservableCollection(Of Kunde)
    Get
        Return _Kunden
    End Get
    Set(value As ObservableCollection(Of Kunde))
        _Kunden = value
    End Set
End Property
```

Schließlich sorgen wir in der Konstruktor-Methode der Klasse, die beim Anzeigen automatisch ausgeführt wird, dass die XAML-Definition umgesetzt, der **DataContext** auf die aktuelle Code behind-Klasse gesetzt und die Eigenschaft **_Kunden** mit dem Ergebnis der Funktion **GetKunden** gefüllt wird:

```
Public Sub New()
    InitializeComponent()
    DataContext = Me
    _Kunden = GetKunden()
End Sub
```

Fenster mit ListView-Steuerelement

Danach definieren wir den XAML-Code für das Fenster zur Anzeige der Daten im **ListView**-Steuerelement:

```
<Window x:Class="MainWindow" ...
        Title="MainWindow" Height="450" Width="800" WindowStartupLocation="CenterScreen">
    ...
    <Grid>
        ...
        <ListView x:Name="lvw" ItemsSource="{Binding Kunden}">
            <ListView.View>
                <GridView>
                    <GridViewColumn Header="ID" DisplayMemberBinding="{Binding ID}" />
                    <GridViewColumn Header="Firma" DisplayMemberBinding="{Binding Firma}" />
                    <GridViewColumn Header="Vorname" DisplayMemberBinding="{Binding Vorname}" />
                    <GridViewColumn Header="Nachname" DisplayMemberBinding="{Binding Nachname}" />
                    <GridViewColumn Header="Anrede" DisplayMemberBinding="{Binding Anrede}" />
                    <GridViewColumn Header="Strasse" DisplayMemberBinding="{Binding Strasse}" />
                    <GridViewColumn Header="PLZ" DisplayMemberBinding="{Binding PLZ}" />
                    <GridViewColumn Header="Ort" DisplayMemberBinding="{Binding Ort}" />
                    <GridViewColumn Header="Land" DisplayMemberBinding="{Binding Land}" />
                    <GridViewColumn Header="EMail" DisplayMemberBinding="{Binding EMail}" />
                    <GridViewColumn Header="Geburtsdatum"
                        DisplayMemberBinding="{Binding Path=Geburtsdatum, StringFormat='dd.MM.yyyy'}" />
                    <GridViewColumn Header="Alter" DisplayMemberBinding="{Binding Alter}" />
                    <GridViewColumn Header="Newsletter">
                        <GridViewColumn.CellTemplate>
                            <DataTemplate>
                                <CheckBox IsChecked="{Binding Newsletter}" />
                            </DataTemplate>
                        </GridViewColumn.CellTemplate>
                    </GridViewColumn>
                </GridView>
            </ListView.View>
        </ListView>
        <StackPanel Orientation="Horizontal" Grid.Row="1">
            <Button>Button</Button>
        </StackPanel>
    </Grid>
</Window>
```

Das **ListView**-Steuerelement stellen wir mit dem Attribut **ItemsSource** so ein, dass es den Wert der Eigenschaft **Kunden** als Datenquelle verwendet. Diese wird in der Code behind-Klasse wie oben beschrieben definiert. Die einzelnen Spalten im **ListView**-Steuerelement definieren wir als **GridViewColumn**-Elemente. Dabei geben wir mit dem Attribut **Header** die Spaltenüberschrift an und mit **DisplayMemberBinding** das Feld, aus der die Spalte ihre Daten beziehen soll. Hier verwenden wir meist den Wert {**Binding [Feldname]**}, wobei **[Feldname]** durch den jeweiligen Feldnamen ersetzt wird. Bei zwei Feldern gibt es Abweichungen von dieser Vorgehensweise: Beim Feld **Geburtsdatum** verwenden wir den Ausdruck {**Binding Path=Geburtsdatum, StringFormat='dd.MM.yyyy'**}, um den Inhalt des Feldes **Geburtsdatum** entsprechend zu formatieren. Und im Falle des Feldes **Newsletter**, das den Datentyp **Boolean** hat, wollen wir den Inhalt in einem Kontrollkästchen anzeigen. Dazu benötigen wir eine etwas andere Struktur, die ein **CheckBox**-Steuerelement enthält, dessen Attribut **Checked** an das Feld **Newsletter** gebunden ist. Anschließend sieht das Fenster nach dem Starten der Anwendung wie in Bild 1 aus.

Grundlagen zum Filtern, Sortieren und Gruppieren

WPF stellt die Infrastruktur zum Filtern, Sortieren und Gruppieren von Daten, die in einem Steuerelement wie beispielsweise dem **ListView**-Steuerelement angezeigt werden, zur Verfügung. Dazu muss man wissen, dass das jeweilige Listen-Steuerelement nicht direkt an die Daten gebunden wird, die es anzeigen soll, sondern an eine View auf diese Daten. Diese View hat bei der in unserem Beispiel verwendeten Auflistung auf Basis der Schnittstelle **IList** den Typ **ListCollectionView**. Diese **ListCollectionView** bietet verschiedene Eigenschaften, die wir nutzen können – im Folgenden verwenden wir zunächst diese beiden:

- **Refresh**: Aktualisiert die Ansicht mit den aktuell angegebenen Filter-, Sortierungs- und Gruppierungseinstellungen
- **Filter**: Stellt ein, welche Funktion beim Aktualisieren zum Filtern verwendet werden soll

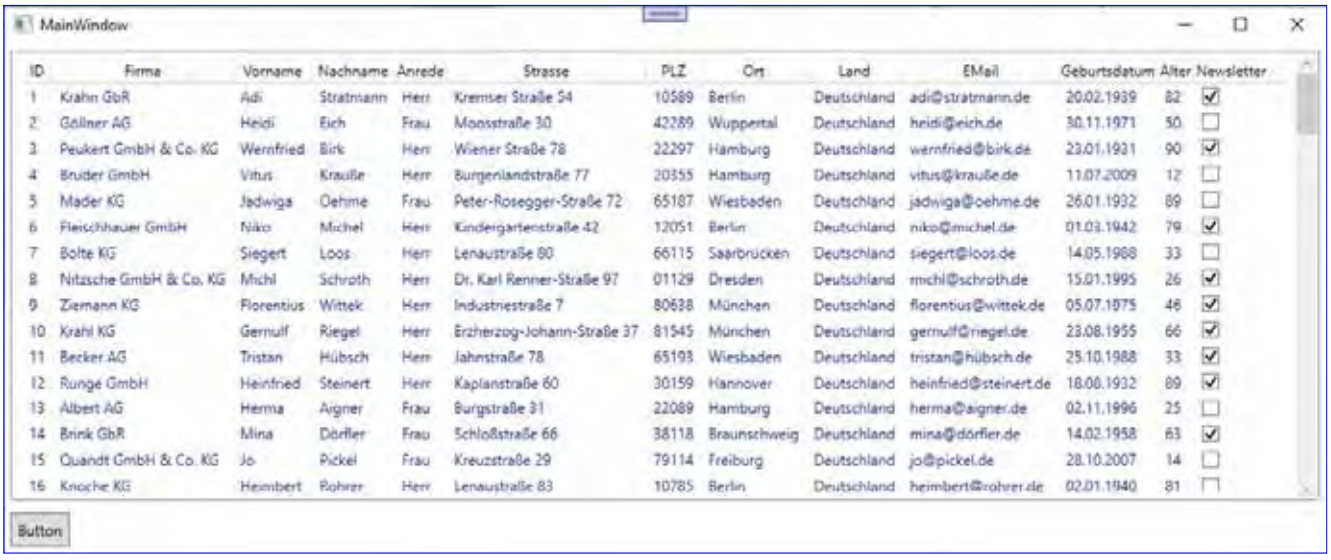


Bild 1: Die Liste der Kunden im **ListView**-Steuerelement

Um die **ListCollectionView** für unsere Zwecke zu nutzen, machen wir diese zunächst per Variable zugreifbar:

```
Private objListCollectionView As ListCollectionView
```

Diese Variable füllen wir gleich in der Konstruktor-Methode des Code behind-Moduls, indem wir die Methode **GetDefaultView** der **CollectionViewSource**-Klasse aufrufen und dieser einen Verweis auf die Liste übergeben, deren **ListCollectionView**-Element wir erhalten wollen:

```
Public Sub New()
    ...
    objListCollectionView = CollectionViewSource.GetDefaultView(Kunden)
```

In der gleichen Methode stellen wir die Eigenschaft **Filter** von **objListCollectionView** ein, und zwar auf die Funktion **KundenFilter**, die wir mit **AddressOf** angeben:

```
    objListCollectionView.Filter = AddressOf KundenFilter
End Sub
```

Bevor wir fortsetzen, fügen wir dem XAML-Code für unser Fenster ein **StackPanel** hinzu, dass ein **Label**-, ein **TextBox**- und ein **Button**-Element enthält:

```
<StackPanel Orientation="Horizontal">
    <Label>Nachname:</Label>
    <TextBox x:Name="txtFilterNachname" Width="100"></TextBox>
    <Button x:Name="btnFiltern" IsDefault="True" Click="btnFiltern_Click">Filtern</Button>
</StackPanel>
```

Das **TextBox**-Element soll den Filterausdruck entgegennehmen, das **Button**-Element soll das Filtern auslösen. Deshalb geben wir dafür eine Ereignismethode an, die wie folgt aussieht und lediglich die **Refresh**-Methode von **objListCollectionView** auslöst:

```
Private Sub btnFiltern_Click(sender As Object, e As RoutedEventArgs)
    objListCollectionView.Refresh()
End Sub
```

Für das Attribut **IsDefault** des **Button**-Elements stellen wir außerdem den Wert **True** ein, damit die Methode hinter dieser Schaltfläche auch beim Betätigen der Eingabetaste ausgelöst wird.

Da wir eine Funktion für die Eigenschaft **Filter** von **objListCollectionView** angegeben haben, wird diese beim Aufruf der **Refresh**-Methode aufgerufen, um die anzuzeigenden Elemente zu ermitteln. Diese Funktion wird tatsächlich für jedes Element der **objListCollectionView** zugrunde liegenden Liste aufgerufen. Das jeweilige

Element wird dabei mit dem Parameter **item** als Parameter übergeben. Die Funktion gibt einen **Boolean**-Wert als Ergebnis zurück, dass angibt, ob der aktuelle Eintrag angezeigt werden soll oder nicht. Wir prüfen also innerhalb der Funktion, ob der aktuelle Eintrag den Filterbedingungen entspricht und geben nur dann den Wert **True** zurück und anderenfalls den Wert **False**. In der folgenden Prozedur prüfen wir, ob das Feld **Nachname** mit dem Ausdruck beginnt, den der Benutzer in das Textfeld **txtFilterNachname** eingegeben hat. Dazu lesen wir zuerst den Wert des Parameters **item** in eine Variable namens **objKunde** mit dem Typ **Kunde** ein. Dann lesen wir den Wert der Eigenschaft **Nachname** des Objekts aus **objKunde** aus. Die **IIf**-Bedingung sorgt dafür, dass im Falle des Wertes eine leere Zeichenkette und anderenfalls der Wert des Feldes **Nachname** in der Variablen **strNachname** landet.

Der Rückgabewert ist das Ergebnis der Prüfung, ob der Inhalt von **strNachname** mit dem Wert aus **txtFilterNachname** beginnt, wobei der Zeichenkettenvergleich mit dem Parameter **StringComparison.OrdinalIgnoreCase** durchgeführt wird – Klein-/Großschreibung wird also nicht berücksichtigt:

```
Private Function KundenFilter(item As Object) As Boolean
    Dim objKunde As Kunde
    Dim strNachname As String
    objKunde = TryCast(item, Kunde)
    strNachname = IIf(objKunde.Nachname = Nothing, "", objKunde.Nachname)
    If Not (strNachname.StartsWith(txtFilterNachname.Text, StringComparison.OrdinalIgnoreCase)) Then
        Return False
    End If
    Return True
End Function
```

Wenn die Bedingung nicht erfüllt ist, liefert die Funktion mit der Anweisung **Return** den Wert **False** zurück und beendet die Funktion. Anderenfalls ist **True** das Ergebnis.

Das Ergebnis sieht wie gewünscht aus (siehe Bild 2). Wenn Sie einen Wert in das Textfeld **txtFilterNachname** eingeben und die Eingabetaste oder die Schaltfläche Filtern betätigen, zeigt das

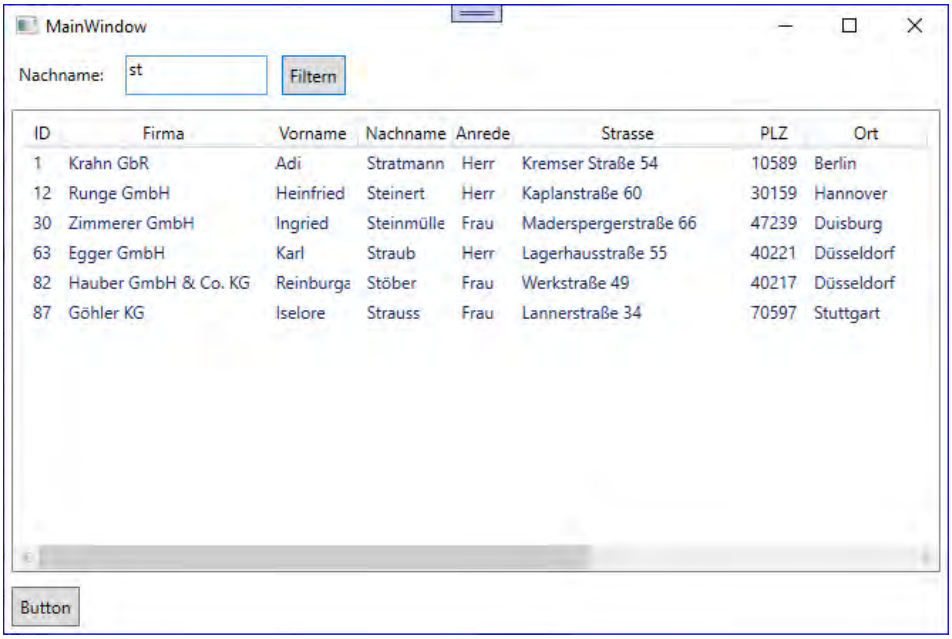


Bild 2: Die gefilterte Liste der Kunden im **ListView**-Steuerelement

Das WPF-Steuerelement CheckBox

»Ja oder Nein – das ist hier die Frage!« Oder doch vielleicht? Wenn es um ein Steuerelement für Ja oder Nein, Wahr oder Falsch, 0 oder -1 geht, ist das Kontrollkästchen die erste Wahl. Oder, wie es in WPF heißt: CheckBox. Mit dem CheckBox-Steuerelement lassen sich üblicherweise zwei Zustände abbilden, die entweder durch einen Haken oder eben durch ein leeres Kästchen dargestellt werden. Ganz nebenbei bemerkt, kann man Kontrollkästchen auch noch für einen dritten Zustand verwenden, nämlich Null oder Nothing – je nach Programmiersprache. Dieser Artikel zeigt, was Sie mit dem CheckBox-Steuerelement von WPF anstellen können.

Das **CheckBox**-Steuerelement ist, was die Möglichkeiten für den Inhalt angeht, eines der einfacheren Steuerelemente von WPF – zumindest unter den Steuerelementen, die Sie an Daten binden können. Warum wir dieses erst im fünften Jahr dieses Magazins untersuchen, kann nur daran liegen, dass es so einfach scheint – dabei gibt es durchaus Eigenschaften, die nicht offensichtlich sind und die wir in diesem Artikel genauer untersuchen wollen.

Das wir das **CheckBox**-Steuerelement nun unter die Lupe nehmen, liegt daran, dass wir in einem anderen Artikel eine wichtige Eigenschaft benötigten, nämlich den dreifachen Zustand: Ist dieser aktiviert, kann das Steuerelement nicht nur die Werte Wahr und Falsch abbilden, sondern auch noch einen dritten Wert, der »nicht Wahr und nicht Falsch« oder auch **Null** (unter C#) oder **Nothing** (unter Visual Basic) lautet. Diesen benötigten wir in einer Suche, wo wir Kunden danach filtern wollten, ob sie einen Newsletter abonniert haben oder nicht. Und natürlich sollte die Suche auch ermöglichen, alle Kunden anzuzeigen, also mit und ohne Newsletterabonnement. Genau dazu brauchten wir den dritten Zustand des Kontrollkästchens. Doch eins nach dem anderen!

CheckBox-Steuerelement anlegen

Das **CheckBox**-Steuerelement legen Sie wie alle anderen Steuerelemente auch durch ein entsprechendes XAML-Element an, das logischerweise **CheckBox** heißt und mit x:Name einen Namen erhält, hier **chkBeispiel**:

```
<Grid>
    <CheckBox x:Name="chkBeispiel"></CheckBox>
</Grid>
```

Ohne weitere Elemente oder Formatierungen erscheint diese allerdings einfach links oben im Fenster (siehe Bild 1).

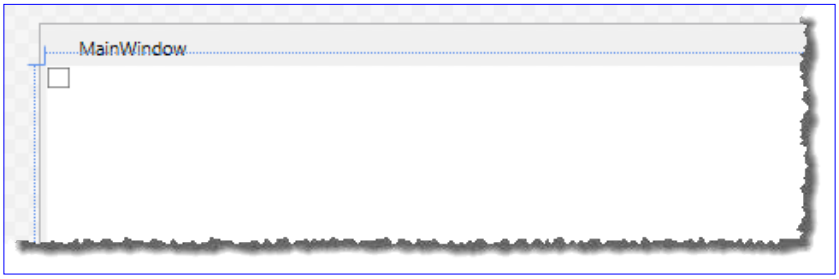


Bild 1: Einfache CheckBox

Beschriftung hinzufügen

Hier fehlt zumindest eine Beschriftung, die wir per **Label**-Element hin-

zufügen. Damit die beiden Elemente nebeneinander erscheinen, fassen wir diese in einem **StackPanel**-Element mit horizontaler Ausrichtung zusammen:

```
<StackPanel Orientation="Horizontal">
    <Label>Beispiel-Checkbox:</Label>
    <CheckBox x:Name="chkBeispiel"></CheckBox>
</StackPanel>
```

Bild 2: CheckBox mit Label

Bild 3: Vertikal ausgerichtete CheckBox mit Label

CheckBox-Elemente ausrichten

Damit erscheint die **CheckBox** allerdings nicht korrekt ausgerichtet (siehe Bild 2). Damit das geschieht, erstellen wir erstens einen Satz von **RowDefinition**-Elementen für das **Grid**, von denen die Höhe der ersten Zeile automatisch an den Inhalt angepasst werden soll. Zweitens stellen für das **CheckBox**-Element das Attribut **VerticalAlignment** auf den Wert **Center** ein:

```
<Grid.RowDefinitions>
    <RowDefinition Height="Auto"></RowDefinition>
    <RowDefinition Height="*"></RowDefinition>
</Grid.RowDefinitions>
<StackPanel Orientation="Horizontal">
    <Label>Beispiel-Checkbox:</Label>
    <CheckBox x:Name="chkBeispiel" VerticalAlignment="Center"></CheckBox>
</StackPanel>
```

Das Ergebnis entspricht schon eher unseren Wünschen (siehe Bild 3).

Mehrere CheckBox-Elemente ausrichten

Wollen Sie jedoch mehrere Kontrollkästchen untereinander darstellen, wobei die Beschriftungen links und die Kontrollkästchen bündig dargestellt werden, benötigen wir eine Erweiterung des **Grid**-Elements um zwei Spalten. Dafür werfen wir das **StackPanel**-Element über Bord und teilen die **Label**- und **CheckBox**-Elemente auf die Zellen des Grids auf:

```
<Grid.RowDefinitions>
    <RowDefinition Height="Auto"></RowDefinition>
    <RowDefinition Height="Auto"></RowDefinition>
    <RowDefinition Height="*"></RowDefinition>
</Grid.RowDefinitions>
<Grid.ColumnDefinitions>
    <ColumnDefinition Width="Auto"></ColumnDefinition>
```

```
    <ColumnDefinition Width="*"></ColumnDefinition>
</Grid.ColumnDefinitions>
<Label>Beispiel-Checkbox:</Label>
<CheckBox x:Name="chkBeispiel" Grid.Column="1" VerticalAlignment="Center"></CheckBox>
<Label Grid.Row="1">Zweite Checkbox:</Label>
<CheckBox x:Name="chkZwei" Grid.Row="1" Grid.Column="1" VerticalAlignment="Center"></CheckBox>
```

Damit erhalten wir dann die ordentlich ausgerichteten **Label**- und **CheckBox**-Elemente aus Bild 4.

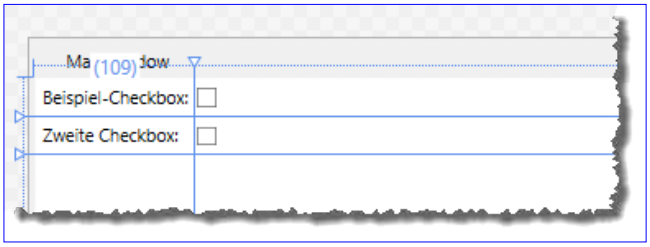


Bild 4: Mehrere ausgerichtete **CheckBox**-Elemente

CheckBox-Elemente links vom Label darstellen

Natürlich können Sie diese auch andersherum darstellen, also die **CheckBox**-Elemente auf der linken Seite platzieren. Auch das ist durchaus üblich, zumindest, wenn nur **CheckBox**-Elemente verwendet werden.

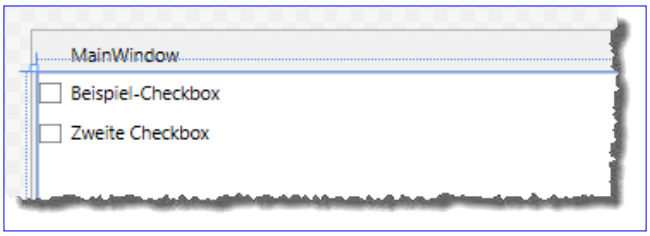


Bild 5: **CheckBox**-Elemente auf der linken Seite

Das hätte auch den Vorteil, dass die **CheckBox**- und die **Label**-Elemente auch bei Verwendung von **StackPanel**-Elementen immer korrekt ausgerichtet sind:

```
<StackPanel Orientation="Vertical">
    <StackPanel Orientation="Horizontal">
        <CheckBox x:Name="chkBeispiel" VerticalAlignment="Center"></CheckBox>
        <Label>Beispiel-Checkbox</Label>
    </StackPanel>
    <StackPanel Orientation="Horizontal">
        <CheckBox x:Name="chkZwei" VerticalAlignment="Center"></CheckBox>
        <Label>Zweite Checkbox</Label>
    </StackPanel>
</StackPanel>
```

Das Ergebnis finden Sie in Bild 5.

Das gelingt allerdings auch noch einfacher, denn wir können die Beschriftung in diesem Fall auch als Inhalt des CheckBox-Elements angeben. Der Code dazu sieht so aus:

```
<StackPanel Orientation="Vertical">
    <CheckBox x:Name="chkBeispiel">Beispiel-CheckBox</CheckBox>
    <CheckBox x:Name="chkZwei">Zweite Checkbox</CheckBox>
</StackPanel>
```

Verschiedene Zustände der CheckBox per XAML einstellen

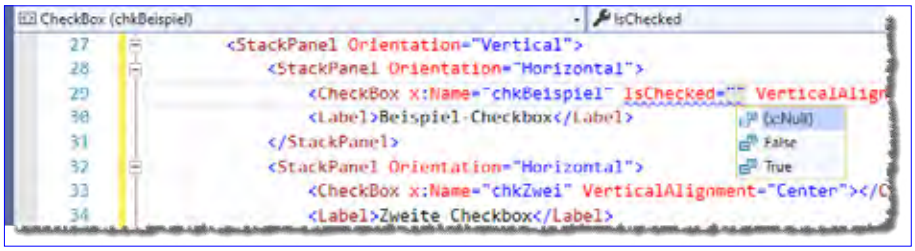


Bild 6: Werte für das Attribut **IsChecked**

Wie weiter oben erwähnt, kann das **CheckBox**-Steuerelement drei verschiedene Status einnehmen. Diese legen Sie im XAML-Code mit dem Attribut **IsChecked** fest. Die drei möglichen Werte lauten dort:

- **True**: Wahr, Ja, -1
- **False**: Falsch, Nein, 0
- **{x:Null}**: Kein Wert

Bild 6 zeigt die Auswahl der verschiedenen Werte per IntelliSense.

Der folgende XAML-Code definiert **CheckBox**-Elemente mit den drei verfügbaren Werten für das Attribut **IsChecked**:

```
<CheckBox x:Name="chkChecked" IsChecked="True" VerticalAlignment="Center"></CheckBox>
<Label>IsChecked = True</Label>
...
<CheckBox x:Name="chkNotChecked" IsChecked="False" VerticalAlignment="Center"></CheckBox>
<Label>IsChecked = False</Label>
...
<CheckBox x:Name="chkNull" IsChecked="{x:Null}" VerticalAlignment="Center"></CheckBox>
<Label>IsChecked = {x:Null}</Label>
```

Diese Elemente sehen nun wie in Bild 7 aus.

Dreifachen Zustand aktivieren

Wenn wir das Projekt zum Debuggen starten, finden wir die drei verschiedenen Zustände wie in der Abbildung vor.

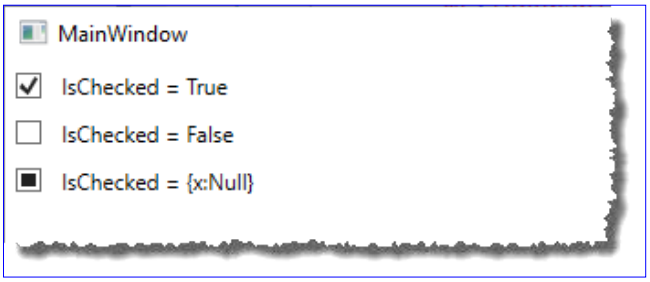


Bild 7: Verschiedene Zustände des **CheckBox**-Elements

Klicken Sie allerdings mehrfach auf eines der **CheckBox**-Elemente, stellen Sie schnell fest, dass Sie immer nur zwischen den beiden Zuständen **Wahr** und **Falsch** wechseln können. Der Zustand **{x:Null}** ist nicht er-

reichbar. Das liegt daran, dass wir das Attribut **IsThreeState** nicht auf den Wert **True** eingestellt haben. Wenn wir das für eines der Elemente nachholen, können Sie zwischen allen drei Zuständen wechseln:

```
<CheckBox x:Name="chkNull" IsChecked="{x:Null}" IsThreeState="True" ...></CheckBox>
```

Der Wert des Attributs **IsThreeState** wirkt sich allerdings nur auf das Anklicken des **CheckBox**-Steuerelements aus. Per Visual Basic können Sie die Eigenschaft durch Einstellen der Eigenschaft **IsChecked** auf den Wert Nothing auf den dritten Zustand einstellen.

CheckBox per Visual Basic abfragen

Um den Wert eines **CheckBox**-Elements abzufragen, legen wir ein Kontrollkästchen namens **chkLesen** an und eine Schaltfläche namens **btnCheckBoxWert**. Diese soll beim Anklicken eine Methode auslösen, die den aktuellen Wert des **CheckBox**-Elements ausgibt:

```
<CheckBox x:Name="chkLesen" IsThreeState="True">Checkbox</CheckBox>
<Button x:Name="btnCheckBoxLesen" Click="btnCheckBoxLesen_Click">Wert ausgeben</Button>
```

Die Methode zeigt ein Meldungsfenster an, das einen Text und den Zustand des **CheckBox**-Elements anzeigt. Dabei ist wichtig, dass das **CheckBox**-Steuerelement keine Eigenschaft namens **Value** aufweist wie viele andere Steuerelemente. Stattdessen verwenden wir die Eigenschaft **IsChecked**, die wir ja auch schon im XAML-Code zum Einstellen des Zustands genutzt haben:

```
Private Sub btnCheckBoxLesen_Click(sender As Object, e As RoutedEventArgs)
    MessageBox.Show("Die CheckBox hat den Wert: " + chkLesen.IsChecked.ToString)
End Sub
```

Dies liefert für ein angeklicktes Kontrollkästchen den Wert **True**, für ein leeres Kontrollkästchen den Wert **False** und für den leeren Zustand eine leere Zeichenfolge. Das ist natürlich nicht der tatsächliche Wert, sondern nur der mit **ToString** in eine Zeichenkette konvertierte Wert.

Den tatsächlichen Wert finden wir beim Debuggen heraus, wenn wir einen Haltepunkt in die Zeile mit der **MessageBox** setzen und den Wert von **chkLesen.IsChecked** prüfen. Dieser hat für ein **CheckBox**-Steuerelement ohne Wert den Wert **Nothing**. Also können wir genauer wie folgt unterscheiden:

```
Private Sub btnCheckBoxLesen_Click(sender As Object, e As RoutedEventArgs)
    If chkLesen.IsChecked Is Nothing Then
        MessageBox.Show("Die Checkbox ist leer.")
    Else
        MessageBox.Show("Die CheckBox hat den Wert: " & chkLesen.IsChecked.ToString)
    End If
End Sub
```

Optionsgruppen mit GroupBox und RadioButtons

Wer von Access kommt, hat vermutlich hier und da Optionsgruppen genutzt. Ein solches Steuerelement bietet WPF nicht direkt an. Allerdings gibt es Elemente, mit dem wir Optionsgruppen nachbauen können. Dabei handelt es sich um das GroupBox- und RadioButton-Steuerelement. Wie Sie damit Optionsgruppen mit Optionen nachbilden, zeigen wir in diesem Artikel.

Einfache Optionsgruppe mit dem GroupBox-Element

Wenn wir von Optionsgruppen reden, benötigen wir auch noch Optionsfelder. Diese bilden wir unter WPF mit dem **RadioButton**-Element ab. In der Definition einer Optionsgruppe können wir nicht einfach die benötigten **RadioButton**-Elemente unter dem **GroupBox**-Element anordnen. Der Grund dafür ist, dass das **GroupBox**-Element nur ein weiteres Element direkt aufnehmen kann – also beispielsweise ein **StackPanel**-Element, um die Optionsfelder wie in Bild 1 nebeneinander oder untereinander anzuordnen.

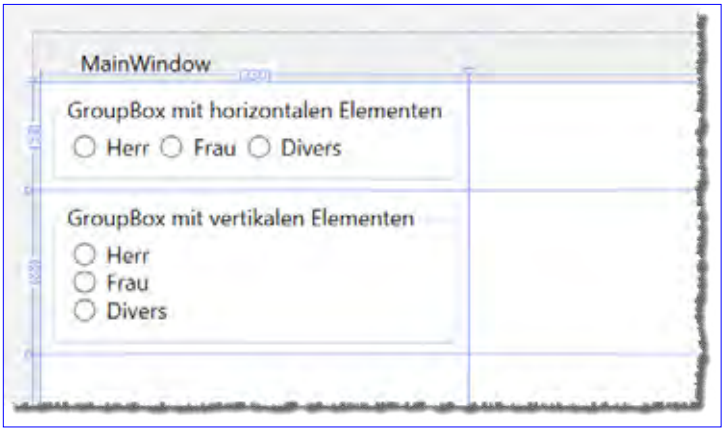


Bild 1: Einfache **GroupBox**-Beispiele

Die **GroupBox** mit horizontal angeordneten Elementen erhalten wir mit dem folgenden Code:

```
<GroupBox Header="Ein GroupBox-Element">
    <StackPanel Orientation="Horizontal">
        <RadioButton>Herr</RadioButton>
        <RadioButton>Frau</RadioButton>
        <RadioButton>Divers</RadioButton>
    </StackPanel>
</GroupBox>
```

Damit die Elemente den Abstand wie in der Abbildung angezeigt erhalten, haben wir noch das Attribut **Margin** für alle Elemente des Typs **RadioButton** angepasst – und gleich auch noch den inneren und äußeren Abstand des **GroupBox**-Elements:

```
<Window.Resources>
    <Style TargetType="GroupBox">
        <Setter Property="Margin" Value="5"></Setter>
        <Setter Property="Padding" Value="5"></Setter>
    </Style>
    <Style TargetType="RadioButton">
```

```
        <Setter Property="Margin" Va-
lue="0,0,5,0"></Setter>
    </Style>
</Window.Resources>
```

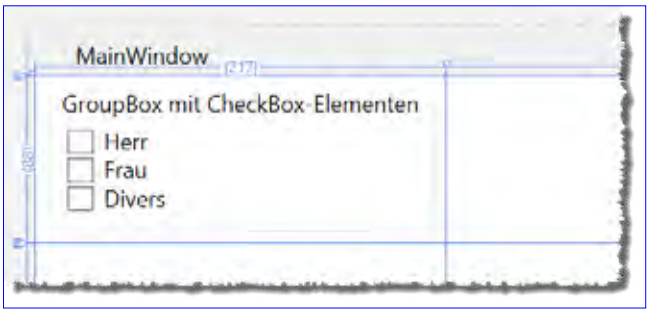


Bild 3: **GroupBox** mit **CheckBox**-Elementen

Die Definition der **GroupBox** mit vertikal angeordneten Elementen lautet wie folgt, wobei der Unterschied im Wert für das Attribut **Orientation** des **StackPanel**-Elements liegt:

```
<GroupBox Header="Noch ein GroupBox-Element" Grid.Row="1">
    <StackPanel Orientation="Vertical">
        <RadioButton>Herr</RadioButton>
        <RadioButton>Frau</RadioButton>
        <RadioButton>Divers</RadioButton>
    </StackPanel>
</GroupBox>
```

Wie wir aus den Beispielen erkennen können, bietet das **GroupBox**-Element primär einen Rahmen und die Möglichkeit, oben in diesem Rahmen eine Bezeichnung einzufügen – und zwar mit dem Attribut **Header**.

GroupBox mit CheckBox-Elementen

Alternativ zu den **RadioButton**-Elementen können Sie auch **CheckBox**-Elemente nutzen:

```
<GroupBox Header="GroupBox mit CheckBox-Elementen" Grid.Row="2">
    <StackPanel Orientation="Vertical">
        <CheckBox>Herr</CheckBox>
        <CheckBox>Frau</CheckBox>
        <CheckBox>Divers</CheckBox>
    </StackPanel>
</GroupBox>
```

Diese erscheinen wie in Bild 3.

Header der GroupBox gestalten

Das **Header**-Element der **GroupBox** können Sie im Gegensatz zu dem der Optionsgruppe aus Access beliebig gestalten. Dazu nehmen Sie das Attribut aus dem eigentlichen Element heraus und formulieren es als Property Element. Hier können Sie dann beispielsweise Attribute defi-

Bild 2: **GroupBox** mit fetter Überschrift

nieren, welche die Schriftart für den Inhalt des Headers festlegen. In folgendem Beispiel haben wir im Element **GroupBox.Header** ein **Label**-Element hinzugefügt und dessen Attribut **FontWeight** auf **Bold** eingestellt:

```
<GroupBox Grid.Row="2">
    <GroupBox.Header>
        <Label FontWeight="Bold">GroupBox mit fetter Überschrift</Label>
    </GroupBox.Header>
    <StackPanel Orientation="Vertical">
        ...
    </StackPanel>
</GroupBox>
```

Das Ergebnis sehen Sie in Bild 2.

Und Sie können nicht nur die Schriftart der Überschrift der Optionsgruppe nach Belieben gestalten. Sie können dem **Header**-Element auch noch beliebige andere Elemente unterordnen – so können Sie beispielsweise auch noch ein Bild hinzufügen. Das Bild hinterlegen wir in einem neuen Ordner im Projekt namens **images**. Der Code zum Anzeigen des dort hinzugefügten Bildes sieht wie folgt aus:

```
<GroupBox Grid.Row="2">
    <GroupBox.Header>
        <StackPanel Orientation="Horizontal">
            <Image Source="images/users_meeting.png"></Image>
            <Label FontWeight="Bold">GroupBox mit fetter Überschrift</Label>
        </StackPanel>
    </GroupBox.Header>
    <StackPanel Orientation="Vertical">
    ...
    </StackPanel>
</GroupBox>
```

Das Property Element **GroupBox.Header** kann ebenfalls nur ein Element in der ersten Ebene enthalten, daher fassen wir das **Image**- und das **Label**-Element in einem **StackPanel**-Element mit horizontaler Orientierung zusammen. Das **Header**-Element mit Bild und Text sehen Sie in Bild 4.

Bild 4: **GroupBox** mit Bild im **Header**-Element

Wert einer GroupBox auslesen

Die bisherigen Ausführungen haben gezeigt, wie Sie eine Optionsgruppe als **GroupBox** mit **RadioButton**-

Optionsgruppe an Boolean binden

Im Artikel »Optionsgruppen mit GroupBox und RadioButtons« haben Sie erfahren, wie Sie Optionsgruppen mit dem GroupBox-Steuerelement und RadioButton-Steuerelementen nachbilden. In diesem Artikel haben wir uns noch nicht darum gekümmert, diesen Optionsgruppen-Nachbau an Daten zu binden – sprich beispielsweise an eine Eigenschaft im Code behind-Modul des Fensters mit der Optionsgruppe. Da dies nicht so trivial ist wie das Binden von Steuerelementen wie TextBox oder anderen, widmen wir diesem Thema einen eigenen Artikel.

GroupBox-Element an Daten binden

In unserem Beispiel wollen wir ein **Boolean**-Feld eines Objekts wie in diesem Fall das Feld **Newsletter**, das angibt, ob ein Kunde den Newsletter erhalten soll, an zwei **GroupBox**-Elemente binden. Dazu erstellen wir im Code behind-Modul des Fensters **MainWindow.xaml** ein **Kunde**-Objekt mit wenigen Eigenschaften und weisen es der Eigenschaft **Kunde** des Moduls zu:

```
Public Property Kunde As Kunde

Public Sub New()
    InitializeComponent()
    DataContext = Me
    _Kunde = New Kunde() With {.ID = 1, .Vorname = "Adi", .Nachname = "Stratmann", .Newsletter = False}
End Sub
```

Das Fenster definieren wir im XAML-Code so, dass es zwei **RadioButton**-Elemente zur Anzeige des **Boolean**-Feldes **Newsletter** nutzt. Damit nur eines davon markiert ist, fassen wir die **RadioButton**-Elemente durch Vergabe des gleichen Wertes **NewsletterGroup** für das Attribut **GroupName** zusammen (siehe Bild 1). Für die Bindung nutzen wir das Attribut **IsChecked**. Dieses binden für wir das **Ja**-Feld einfach an die Eigenschaft **Kunde.Newsletter**. Für das **Nein**-Feld müssen wir es allerdings einen **ValueConverter** hinzunehmen:

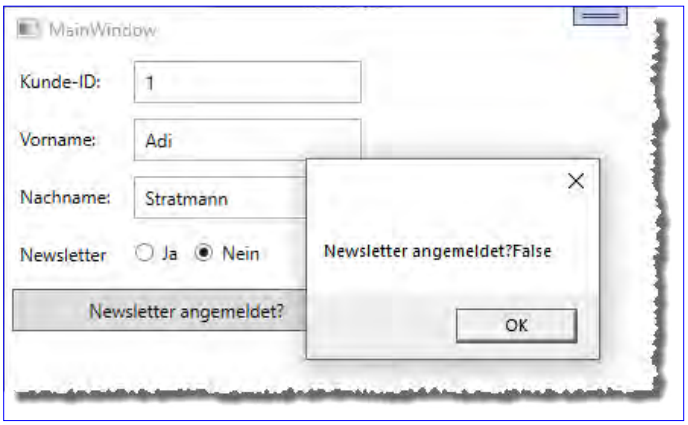


Bild 1: Wert der Eigenschaft auslesen

```
<Window x:Class="MainWindow" ... Title="MainWindow" Height="450" Width="800">
    <Grid>
        ...
        <Label x:Name="lblNewsletter" ...>Newsletter</Label>
        <StackPanel ... Orientation="Horizontal">
            <RadioButton GroupName="NewsletterGroup"
```

Markierung bei Fokuserhalt der TextBox

Im Artikel »Das TextBox-Steuerelement« haben wir die Grundlagen zum TextBox-Steuerelement beschrieben. Dort haben wir noch längst nicht alle Möglichkeiten behandelt, die es bei der Programmierung von TextBox-Steuerelementen unter WPF gibt. Im vorliegenden Artikel schauen wir uns an, wie Sie das Verhalten einer TextBox beim Fokuserhalt steuern können, genau genommen: Wie erreiche ich, dass die Einfügemarke an einer bestimmten Stelle positioniert wird beziehungsweise der komplette Inhalt markiert wird? Und das nicht nur für den Fokuserhalt durch die Tabulator- oder Eingabetaste, sondern auch beim Anklicken mit der Maustaste.

Einfügemarke beim Fokuserhalt an bestimmter Position platzieren

Die folgenden Definitionen von **TextBox**-Elementen sollen die Einfügemarke an verschiedenen Positionen platzieren, wenn das Element den Fokus erhält. Das erste Element soll die Einfügemarke am Start des Textes platzieren. Dazu nutzen wir das Ereignis **GotFocus**:

```
<Label>Einfügemarke vorn:</Label>
<TextBox x:Name="txtCursorStart" GotFocus="txtCursorStart_GotFocus">Dies ist ein Beispieltext.</TextBox>
```

Dafür hinterlegen wir die folgende Ereignismethode:

```
Private Sub txtCursorStart_GotFocus(sender As Object, e As RoutedEventArgs)
    txtCursorStart.Select(0, 0)
End Sub
```

Diese verwendet die **Select**-Methode, um mit dem ersten Parameter die Startposition und mit der zweiten die Länge der Markierung festzulegen. Der Wert **0** für den ersten Parameter fügt die Einfügemarke ganz vorn ein und der Wert **0** für den zweiten Parameter sorgt dafür, dass die Einfügemarke nicht zu einer Markierung wird.

Für die Anzeige der Einfügemarke ganz hinten im Textfeld fügen wir ein weiteres Textfeld hinzu. Für dieses definieren wir das gleiche Ereignis:

```
<Label >Einfügemarke hinten:</Label>
<TextBox x:Name="txtCursorEnd" GotFocus="txtCursorEnd_GotFocus">Dies ist ein Beispieltext.</TextBox>
```

Die Methode, die durch dieses Ereignis ausgelöst wird, ermittelt für den ersten Parameter der **Select**-Methode jedoch mit der **Length**-Eigenschaft die Länge des im **TextBox**-Element enthaltenen und mit der Eigenschaft **Text** gelieferten Text. Der zweite Parameter hat wie im ersten Beispiel den Wert **0**:

```
Private Sub txtCursorEnd_GotFocus(sender As Object, e As RoutedEventArgs)
    txtCursorEnd.Select(txtCursorEnd.Text.Length, 0)
```

```
End Sub
```

Im dritten Textfeld wollen wir den vollständigen Text markieren, wenn das Textfeld den Fokus erhält:

```
<Label>Alles markieren:</Label>
<TextBox x:Name="txtCursorAll" GotFocus="txtCursorAll_GotFocus">Dies ist ein Beispieltext.</TextBox>
```

Wir könnten auch hier die **Select**-Methode verwenden und als Startposition der Markierung den Wert **0** und als Endposition die Länge des zu markierenden Textes übergeben. Eleganter ist in diesem Fall jedoch der Einsatz der **SelectAll**-Methode:

```
Private Sub txtCursorAll_GotFocus(sender As Object, e As RoutedEventArgs)
    txtCursorAll.SelectAll()
End Sub
```

All diese Möglichkeiten funktionieren gut, wenn der Benutzer den Fokus mit der Tabulator-Taste oder mit der Eingabe-Taste verschiebt (siehe Bild 1).

Sobald er jedoch mit der Maus in eines der Textfelder klickt, erscheint die Einfügemarke genau an der Stelle des Mausklicks. Beim Anklicken der dritten Schaltfläche, die den ganzen Text markieren sollte, blinkt sogar kurz die vollständige Markierung auf, bevor die Einfügemarke an der Mausposition erscheint.

In manchen Fällen wünschen wir uns jedoch auch beim Anklicken eines **TextBox**-Elements mit der Maus, dass dieses komplett markiert wird. Für diesen Fall müssen wir uns also eine alternative Möglichkeit einfallen lassen. Aber benötigen wir ein anderes Ereignis? Nein, wir ändern lediglich den Code des Ereignisses:

```
Private Sub txtCursorAll_GotFocus(sender As Object, e As RoutedEventArgs)
    Dim txt As TextBox
    txt = sender
    txt.Dispatcher.BeginInvoke(New Action(Sub() txt.SelectAll()))
End Sub
```

Was geschieht hier genau? Es handelt sich um ein Timing-Problem. **GotFocus** wird bereits ausgelöst, wenn Sie die Maustaste im **TextBox**-Steuerelement herunterdrücken. Dann markiert der Code den kompletten Inhalt und durch das Loslassen der Maustaste wird dann die Einfügemarke dort platziert, wo dies geschieht.

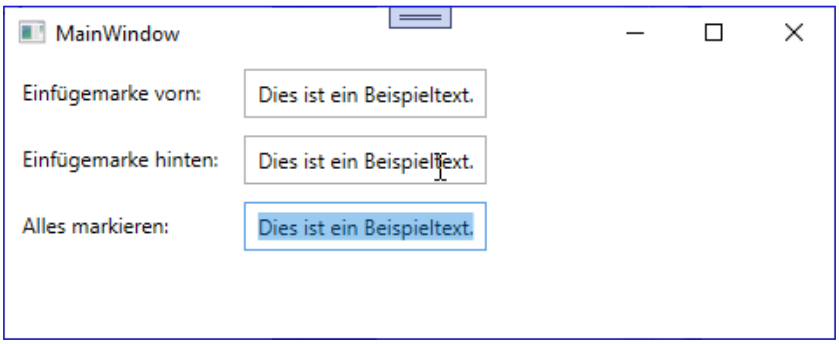


Bild 1: Markieren von Text

AfterUpdate-Ereignis für TextBox-Elemente

Das TextBox-Steuerelement bietet einige Ereignisse, die sich rund um die Dateneingabe drehen. Eines davon heißt TextChanged – dieses haben wir bereits im Artikel »Das TextBox-Steuerelement« vorgestellt. Während dieses Ereignis bei Eingabe jedes einzelnen Zeichens feuert, benötigen wir auch noch ein Ereignis, das uns mitteilt, wann der Benutzer die Eingabe in ein TextBox-Element abgeschlossen hat – beispielsweise, um nach der Eingabe der Nummer eines Datensatzes in einer Datensatznavigation direkt zum gewünschten Datensatz zu springen oder auch eine Suche nach dem eingegebenen Begriff zu starten. In diesem Artikel zeigen wir, wie das gelingt.

Im Artikel **Das TextBox-Steuerelement** (**www.datenbankentwickler.net/215**) haben wir bereits das Ereignis **TextChanged** vorgestellt. Dieses Ereignis ist praktisch, aber in manchen Fällen benötigen wir doch noch weitere Möglichkeiten zum Programmieren von Textfeldern. Das Ereignis **TextChanged** feuert nach jeder Änderung des enthaltenen Texts, also auch dann, wenn der Benutzer nur ein Zeichen hinzugefügt oder entfernt hat.

Wenn wir aber beispielsweise wissen möchten, wann die Änderung abgeschlossen ist – also beispielsweise durch Betätigen der Eingabetaste oder der Tabulator-Taste oder durch Verschieben des Fokus auf ein anderes Steuerelement mit der Maus –, dann benötigen wir andere Ereignisse.

Ereignis beim Beenden der Eingabe

Was wir suchen, ist ein Pendant zum Ereignis **Nach Aktualisierung** beziehungsweise **AfterUpdate** von Textfeldern unter Microsoft Access.

Im ersten Näherungsschritt wollen wir das Betätigen der Eingabe- oder Tabulatortaste bei der Eingabe in das Textfeld abfangen. Dazu schauen wir uns zunächst an, wann genau beim Betätigen einer dieser Tasten unter Access das **AfterUpdate**-Ereignis ausgelöst wird. Dabei stellen wir fest, dass dies bereits beim Herunterdrücken der Taste geschieht. Also bauen wir genau dieses Verhalten nach – und zwar mit der **KeyDown**-Methode.

Dazu legen wir folgendes **TextBox**-Element an und hinterlegen direkt das Attribut **KeyDown** mit der Angabe der auszuführenden Ereignismethode:

```
<TextBox x:Name="txtAfterUpdate" KeyDown="txtAfterUpdate_KeyDown"></TextBox>
```

Diese soll die folgende Ereignismethode auslösen:

```
Private Sub txtAfterUpdate_KeyDown(sender As Object, e As KeyEventArgs)
    Dim txt As TextBox
    txt = sender
    Select Case e.Key
        Case Key.Enter, Key.Tab
```

```
            MessageBox.Show("Der Text lautet '" & txt.Text & "'")
    End Select
End Sub
```

Bild 1: Ausgabe einer Meldung beim Verlassen des Feldes

Die Methode referenziert das mit dem Parameter **sender** übergebene auslösende Steuerelement mit der Variablen **txt** und prüft dann, ob das mit dem Parameter **e** gelieferte Objekt des Typs **KeyEventArgs** für die Eigenschaft **Key** entweder den Wert **Key.Enter** oder **Key.Tab** liefert. Ist das der Fall, soll eine Meldung mit dem aktuellen Inhalt des Textfeldes ausgegeben werden.

Das Ergebnis ist für den ersten Zwischenschritt schon nicht schlecht: Wenn wir das Textfeld mit der Eingabe- oder Tabulatortaste verlassen, wird das Ereignis ausgelöst und gibt den aktuellen Text aus (siehe Bild 1).

AcceptsReturn und AcceptsTab

Es gibt jedoch zwei Attribute namens **AcceptsReturn** und **AcceptsTab**, die angeben, ob das Betätigen der Eingabe- beziehungsweise der Tabulatortaste sich auf den Inhalt des Textfeldes auswirkt oder ob diese als Abschluss der Eingabe interpretiert werden sollen.

Stellen Sie beispielsweise **AcceptsReturn** auf **True** ein, dann löst die Betätigung der Eingabetaste innerhalb des Textfeldes nicht das Ereignis **KeyDown** aus. Gleiches gilt für die Tabulatortaste in Verbindung mit dem Attribut **AcceptsTab**.

Ereignis nur nach Änderung

Nun wollen wir aber dass Ereignis After Update abbilden, was nur dann ausgelöst wird, wenn der Benutzer gegenüber dem vorherigen Zustand auch eine Änderung am Inhalt des Textfeldes durchgeführt hat. Wir müssen also auf irgendeine Weise abgleichen, ob der beim Auslösen der Methode im **TextBox**-Element enthaltene Wert sich von dem unterscheidet, der vor der Änderung vorhanden war.

Alternative: Datenbindung mit UpdateSourceTrigger

Sie können auch einfach eine der Datenbindungseigenschaften von WPF nutzen. In diesem Fall hinterlegen wir in der Code behind-Klasse des Fensters eine private Variable mit einer entsprechenden Property mit Getter und Setter namens **GebundenerText**:

```
Private _GebundenerText As Object

Public Property GebundenerText
    Get
        Return _GebundenerText
    End Get
```

Typumwandlung unter VB.NET

Unter VB.NET gibt es verschiedene Möglichkeiten, den Typ von Variablen umzuwandeln. Dieser Artikel erläutert den Unterschied zwischen impliziter und expliziter Typumwandlung, die Bedeutung von Option Strict, die verschiedenen Typumwandlungsfunktionen wie CBool und die Methoden der Convert-Klasse. Außerdem gibt es noch flexible Umwandlungsfunktionen wie CType, DirectCast oder TryCast – auch diese schauen wir uns genauer an.

Visual Basic.NET bietet sehr viele verschiedene Datentypen. Manchmal müssen Sie Inhalte von einem zum anderen Datentyp konvertieren. Wie das gelingt, zeigen die folgenden Abschnitte.

Implizite Typumwandlung

Die implizite Typumwandlung erfolgt bei der Zuweisung eines Wertes oder eines Objekts zu einer Variablen. Diese Typumwandlung gelingt jedoch nur, wenn der umzuwandelnde Datentyp ein niederwertiger Datentyp ist und der Zieldatentyp ein höherwertiger Datentyp. Das bedeutet zum Beispiel, dass alle Datentypen in den Typ **Object** umgewandelt werden können, da dieser der oberste Datentyp in der Hierarchie der Datentypen ist. Die nachfolgenden Beispiele liefern gültige Zuweisungen, da der Zieldatentyp jeweils **Object** ist:

```
Dim objInteger As Object
objInteger = 10
Dim objKunde As Object
objKunde = New Kunde
```

Für die Zuweisung eines neuen **Object**-Elements zur Objektvariablen des Typs **Kunde** wie in der folgenden Anweisung gelingt das nicht:

```
Dim objKunde2 As Kunde
objKunde2 = New Object
```

Hier erhalten wir eine Fehlermeldung (siehe Bild 1).

Anzeige von ungültigen Zuweisungen per Option aktivieren

Sie können solche Fehler schon vor dem Ausführen ausschließen. Dazu stellen Sie einfach eine spezielle Option namens **Option Strict** auf den Wert **On** ein. Diese finden sie, wenn Sie

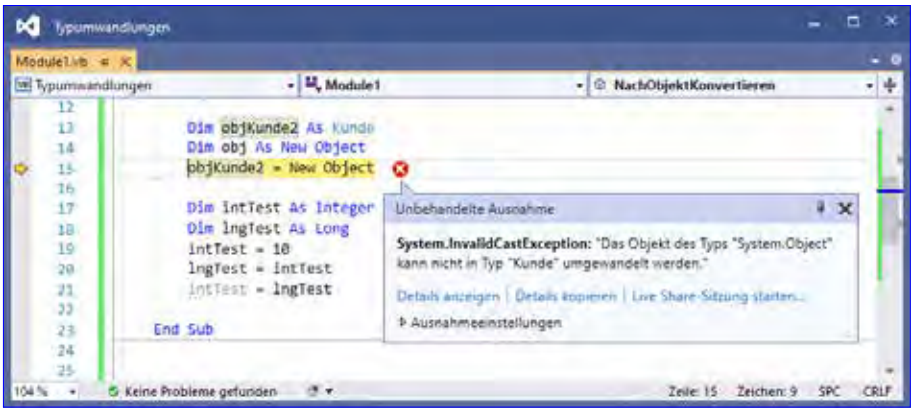


Bild 1: Fehler beim Zuweisen von **Object** zu einem anderen Objekttyp

im Projektmappen-Explorer doppelt auf den Eintrag **My Project** klicken und dort zum Bereich **Kompilieren** wechseln. Hier finden Sie im Bereich **Kompilierungsoptionen** die Eigenschaft **Option Strict**, die wir auf den Wert **On** einstellen (siehe Bild 2).

Damit sorgen wir dafür, dass fehlerhafte Zuweisungen bereits im Codefenster angezeigt werden. Fahren Sie mit der Maus über einen entsprechend markierten Ausdruck, zeigt dieser das Problem an (siehe Bild 3). Hier sehen Sie weiter unten auch eine Markierung einer nicht gültigen Konvertierung von **Long** in **Integer**.

Option Strict als Standard setzen

Standardmäßig ist **Option Strict** für neue Projekte nicht auf den Wert **On** eingestellt, sondern auf **Off**. Wenn Sie wollen, dass diese Einstellung für alle neuen Projekte auf **On** gesetzt wird, öffnen Sie mit **Extras|Optionen** den

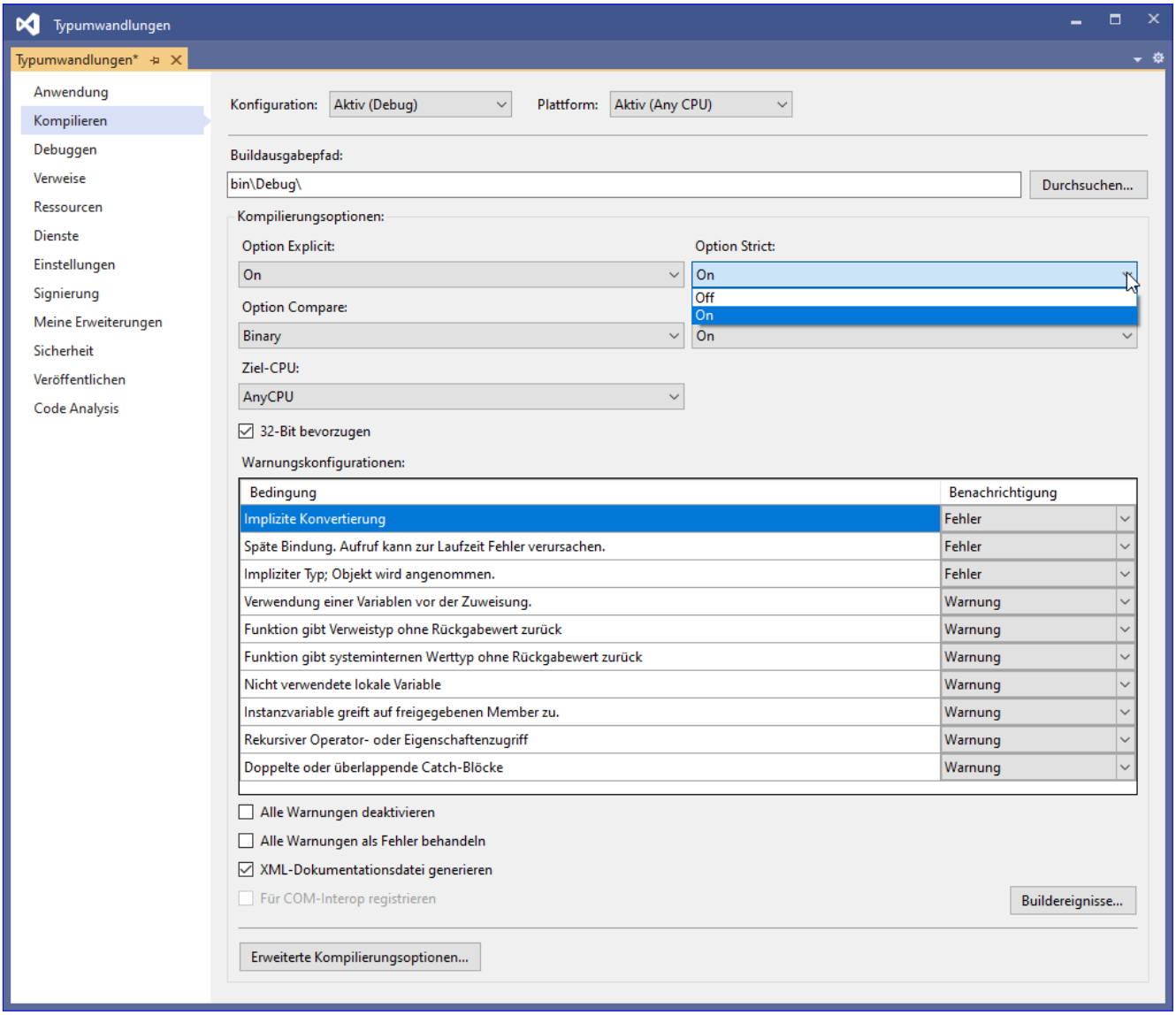


Bild 2: Einstellungen für das Kompilieren

Optionen-Dialog von Visual Studio und navigieren dort zum Bereich **Projekte und Projektmappen|VB-Standard**. Hier stellen Sie die Option **Option Strict** auf **On** ein (siehe Bild 4).

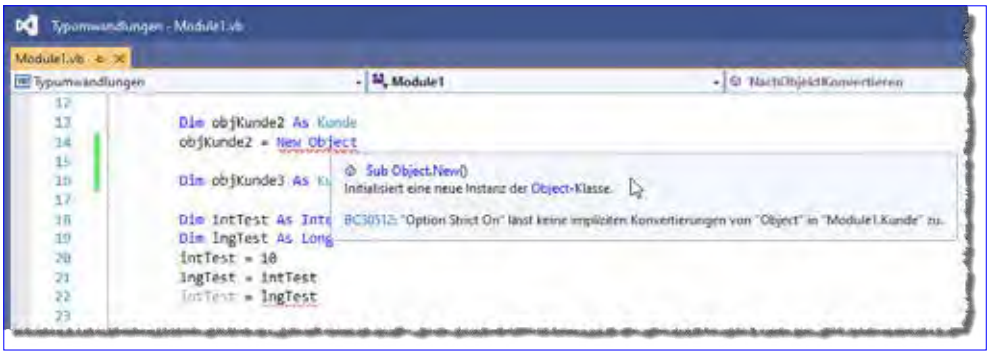


Bild 3: Anzeige nicht erlaubter Konvertierungen

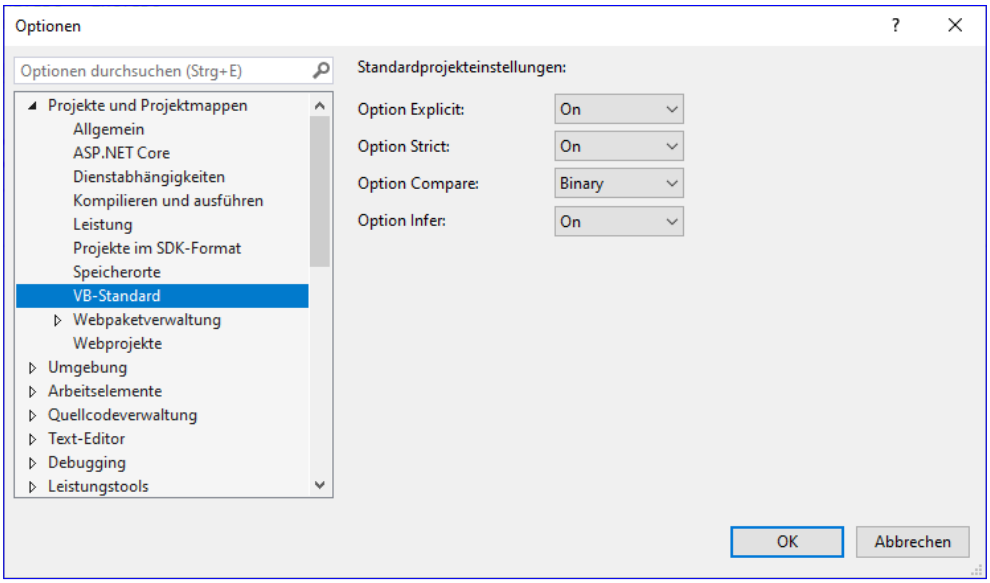


Bild 4: Voreinstellen von **Option Strict** für neue Projekte

Explizite Typkonvertierung

Neben der impliziten gibt es die explizite Typkonvertierung. Dabei verwenden wir eine Funktion, um die Konvertierung von dem einen in den anderen Typ vorzunehmen. Einige dieser Funktionen können Sie zum Konvertieren in spezielle Datentypen verwenden, zum Beispiel **CBool**. Damit konvertieren Sie den als Parameter angegebenen Ausdruck in ein Objekt des Typs **Boolean**:

AUSGABE 05/2021

DATENBANK ENTWICKLER

MAGAZIN FÜR DIE DATENBANKENTWICKLUNG MIT VISUAL STUDIO FÜR DESKTOP, WEB UND CO.

TOP-THEMEN:

WPF	Navigieren mit CollectionViewSource	SEITE 3
VISUAL STUDIO	Versionsverwaltung mit Git	SEITE 13
DATENZUGRIFF	Beispieldaten generieren mit Bogus	SEITE 25
LÖSUNGEN	Seminarverwaltung Teil I-III	SEITE 47

André Minhorst Verlag

Navigieren mit CollectionViewSource

Bisher haben wir oft mit Detailansichten von Datensätzen wie beispielsweise für Kunden oder Produkte gearbeitet, die zum Anlegen oder Bearbeiten eines einzelnen Datensatzes geeignet waren. Von Access kennen Sie die Möglichkeit, mit den Navigationsschaltflächen auch in solchen Detailformularen zu navigieren und von einem zum anderen Datensatz zu wechseln, ohne zwischendurch ein Übersichtsformular zu benötigen. In diesem Artikel wollen wir zeigen, wie Sie das unter WPF so abbilden können, wie es auch unter Access möglich ist. Dabei nutzen wir die Möglichkeiten der CollectionViewSource.

Ziel des Artikels

In diesem Artikel wollen wir einer Seite, die eine Detailansicht eines Datensatzes – hier eines Kunden – anzeigt, Navigationsschaltflächen wie von den Formularen von Access bekannt hinzufügen. Die WPF-Seite besteht dabei grob aus einem **Grid**-Element, das wiederum zwei weitere **Grid**-Element aufnimmt. Es definiert außerdem im Element **Window.Resources** ein **CollectionViewSource**-Element, das wir im Code behind mit den anzuzeigenden Daten füllen:

```
<Window x:Class="MainWindow" ... Title="MainWindow" Height="450" Width="800">
    <Window.Resources>
        ...
        <CollectionViewSource x:Key="kundeViewSource"></CollectionViewSource>
    </Window.Resources>
```

Danach folgt die Definition des übergeordneten **Grid**-Elements:

```
    <Grid>
        <Grid.RowDefinitions>
            <RowDefinition Height="Auto"></RowDefinition>
            <RowDefinition Height="Auto"></RowDefinition>
            <RowDefinition Height="*"></RowDefinition>
        </Grid.RowDefinitions>
```

Das erste untergeordnete **Grid**-Element erhält als **DataContext** das als statische Ressource hinterlegte **CollectionViewSource**-Element sowie einige Zeilen für die Anzeige der einzelnen Felder und Spalten zur Aufteilung von Bezeichnungsfeldern und gebundenen Steuerelementen:

```
        <Grid DataContext="{StaticResource kundeViewSource}">
            <Grid.RowDefinitions>
                <RowDefinition Height="Auto"></RowDefinition>
                ...
```

```
        <RowDefinition Height="*"></RowDefinition>
    </Grid.RowDefinitions>
    <Grid.ColumnDefinitions>
        <ColumnDefinition Width="Auto"></ColumnDefinition>
        <ColumnDefinition Width="Auto"></ColumnDefinition>
        <ColumnDefinition Width="*"></ColumnDefinition>
    </Grid.ColumnDefinitions>
```

Danach folgen die Definitionen für Steuerelemente, die in den einzelnen Zeilen und Spalten angezeigt werden sollen – zum Beispiel für die ID und den Vornamen:

```
    <Label Content="ID:" Grid.Column="0" />
    <TextBox x:Name="txtID" Grid.Column="1" HorizontalAlignment="Left"
        Text="{Binding ID, Mode=TwoWay}" Width="50" IsEnabled="False" BorderBrush="Transparent" />
    <Label Content="Vorname:" Grid.Column="0" Grid.Row="2" />
    <TextBox x:Name="txtVorname" Grid.Column="1" Grid.Row="2" Width="202"
        Text="{Binding Vorname, Mode=TwoWay, ValidatesOnDataErrors=True}" Margin="2,2,0,3" />
    ...
</Grid>
```

Das zweite untergeordnete **Grid**-Element enthält ein **StackPanel**-Element mit den vier Schaltflächen und dem Textfeld zur Anzeige der Position des Datensatzzeigers beziehungsweise zur Eingabe des anzuzeigenden Datensatzes, das wir im folgenden Abschnitt beschreiben.

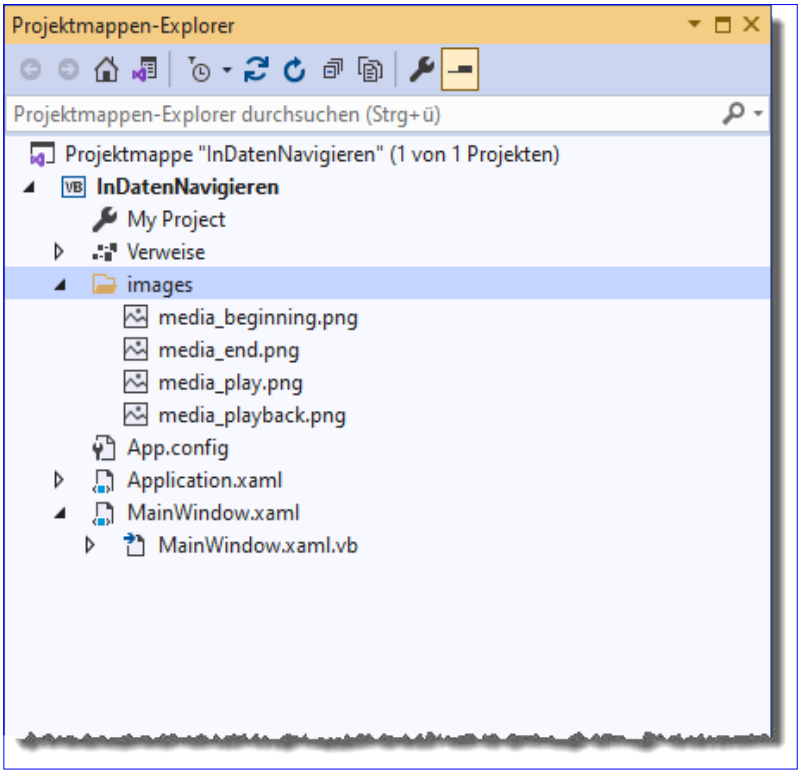


Bild 1: Ordner mit den Bilddateien

Aufbau der Navigationssteuerelemente

Die Navigationsschaltflächen wollen wir genau wie unter Access abbilden. Das heißt, dass wir von links nach rechts die folgenden Steuerelemente hinzufügen wollen:

- Schaltfläche zum Navigieren zum ersten Datensatz
- Schaltfläche zum Navigieren zum vorherigen Datensatz

- Textfeld zur Anzeige der aktuellen Position im Format **x von y**, wobei **y** die Gesamtzahl der angezeigten Datensätze enthält. In dieses Textfeld kann der Benutzer die Position eingeben, zu der er navigieren möchte.
- Schaltfläche zum Navigieren zum nächsten Datensatz
- Schaltfläche zum Navigieren zum letzten Datensatz

Die Steuerelemente organisieren wir in einem **StackPanel**-Steuerelement mit horizontaler Ausrichtung, das wir wie folgt definieren:

Die Bilddateien für die hier definierten **Image**-Elemente finden sie im Beispielprojekt im Ordner **images** (siehe Bild 1).

Der XAML-Code hierfür sieht wie folgt aus:

```
<Grid Grid.Row="1">
    <StackPanel Orientation="Horizontal" Grid.Row="10" Grid.ColumnSpan="4" >
        <Button x:Name="btnFirst" Click="btnFirst_Click" IsEnabled="{Binding FirstOrPreviousEnabled}">
            <Image Source="images/media_beginning.png" Width="24" Height="24"></Image>
        </Button>
        <Button x:Name="btnPrevious" Click="btnPrevious_Click"
            IsEnabled="{Binding FirstOrPreviousEnabled, UpdateSourceTrigger=PropertyChanged}">
            <Image Source="images/media_playback.png" Width="24" Height="24"></Image>
        </Button>
        <TextBox x:Name="txtNavigation" MinWidth="50"
            Text="{Binding MyCurrentPosition, UpdateSourceTrigger=LostFocus}"
            GotFocus="txtNavigation_GotFocus" KeyDown="txtNavigation_KeyDown"></TextBox>
        <Button x:Name="btnNext" Click="btnNext_Click"
            IsEnabled="{Binding NextOrLastEnabled, UpdateSourceTrigger=PropertyChanged}">
            <Image Source="images/media_play.png" Width="24" Height="24"></Image>
        </Button>
        <Button x:Name="btnLast" Click="btnLast_Click"
            IsEnabled="{Binding NextOrLastEnabled, UpdateSourceTrigger=PropertyChanged}">
            <Image Source="images/media_end.png" Width="24" Height="24"></Image>
        </Button>
    </StackPanel>
</Grid>
```

Für die Schaltflächen finden Sie das Attribut **IsEnabled**, das wir an die Eigenschaften **FirstOrPreviousEnabled** beziehungsweise **NextOrLastEnabled** gebunden haben. Wie diese Eigenschaften definiert sind, beschreiben wir weiter unten.

Der Entwurf des Fensters sieht anschließend wie in Bild 2 aus.

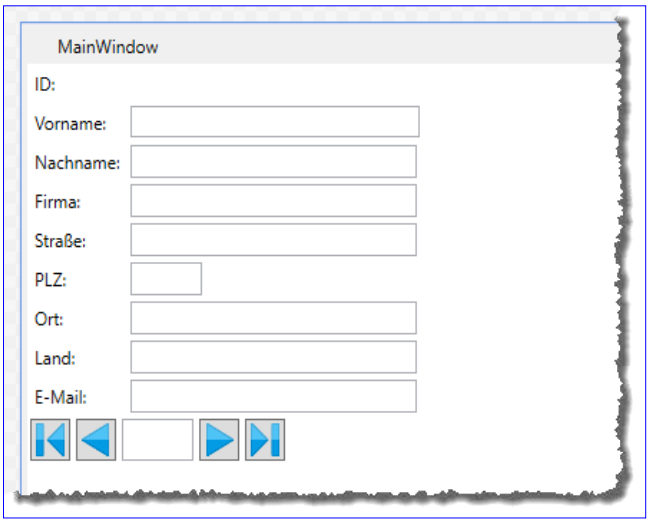


Bild 2: Entwurf des Fensters zur Anzeige der Kunden mit Navigationsschaltflächen

Anzeige der Daten

Damit überhaupt Daten im Fenster angezeigt werden, verwenden wir im Code behind-Modul eine Variable, mit der wir die **CollectionViewSource** referenzieren, die wir im XAML-Code als Ressource definiert haben:

```
Private cvsKunden As CollectionViewSource
```

Diese verbinden wir in der Konstruktor-Methode **New** mit der Ressource und stellen dann die Eigenschaft **Source** des **CollectionViewSource**-Elements auf den Wert der Funktion **GetKunden** ein:

```
Public Sub New()
    InitializeComponent()
    DataContext = Me
    cvsKunden = (Me.FindResource("kundeViewSource"))
    cvsKunden.Source = GetKunden()
End Sub
```

Die Funktion **GetKunden** erstellt eine **ObservableCollection** mit Elementen auf Basis der Klasse **Kunde**. Die Definition dieser Klasse sieht gekürzt wie folgt aus:

```
Partial Public Class Kunde
    Public Property ID As System.Int32
    Public Property Firma As System.String
    Public Property Vorname As System.String
    ...
End Class
```

Die Funktion **GetKunden** fügt mit der **Add**-Methode jeweils ein neues Element hinzu:

```
Private Function GetKunden() As ObservableCollection(Of Kunde)
    Dim Kunden As New ObservableCollection(Of Kunde)
    Kunden.Add(New Kunde() With {.ID = 1, .Firma = "Krahn GbR", .Vorname = "Adi", _
        .Nachname = "Stratmann", .Anrede = "Herr", .Strasse = "Kremser Straße 54", .PLZ = "10589", _
        .Ort = "Berlin", .Land = "Deutschland", .EMail = "adi@stratmann.de", _
        .Geburtsdatum = "20.02.1939", .Alter = 82, .Newsletter = -1})
```

```
    ... 'weitere Add-Aufrufe
    Return Kunden
End Function
```

Schaltflächen zum Navigieren in den Datensätzen

Damit können wir nun die Schaltflächen programmieren. Hier verwenden wir bereits eine Eigenschaft namens **MyCurrentPosition**, mit der wir die Position des Datensatzzeigers verwalten. Wie diese definiert ist, erläutern wir gleich im Anschluss.

Klickt der Benutzer auf die Schaltfläche **btnFirst**, sorgt die folgende Methode mit dem Aufruf von **MoveCurrentToFirst** für die Anzeige des ersten Datensatzes der aktuellen Ansicht des **CollectionViewSource**-Elements.

Außerdem stellt es den Wert der Eigenschaft **MyCurrentPosition** auf den Wert der aktuellen Position plus eins ein. Plus eins deshalb, weil **CurrentPosition** die Position 0-basiert zurückgibt – für die erste Position liefert sie also den Wert **0**:

```
Private Sub btnFirst_Click(sender As Object, e As RoutedEventArgs)
    cvsKunden.View.MoveCurrentToFirst()
    MyCurrentPosition = cvsKunden.View.CurrentPosition + 1
End Sub
```

Auf ähnliche Weise arbeitet die Methode für das **Click**-Ereignis der Schaltfläche **btnPrevious**. Diese verschiebt den Datensatzzeiger mit **MoveCurrentToPrevious** auf den vorherigen Datensatz. Die aktuelle Datensatzposition wird auf die gleiche Weise wie bei **btnFirst** in **MyCurrentPosition** gespeichert:

```
Private Sub btnPrevious_Click(sender As Object, e As RoutedEventArgs)
    cvsKunden.View.MoveCurrentToPrevious()
    MyCurrentPosition = cvsKunden.View.CurrentPosition + 1
End Sub
```

Die **Click**-Methoden für **btnNext** und **btnLast** sehen dementsprechend wie folgt aus:

```
Private Sub btnNext_Click(sender As Object, e As RoutedEventArgs)
    cvsKunden.View.MoveCurrentToNext()
    MyCurrentPosition = cvsKunden.View.CurrentPosition + 1
End Sub

Private Sub btnLast_Click(sender As Object, e As RoutedEventArgs)
    cvsKunden.View.MoveCurrentToLast()
    MyCurrentPosition = cvsKunden.View.CurrentPosition + 1
End Sub
```

Versionsverwaltung mit Git

Wie geht eigentlich Versionsverwaltung mit Visual Studio-Projekten? Wer Software mit Access entwickelt, musste sich darüber nicht viele Gedanken machen: Man kopiert einfach die komplette .accdb-Datei unter einem anderen Namen. Mittlerweile gibt es auch einige Lösungen von Drittherstellern, mit denen die einzelnen Objekte verwaltet werden können. Unter Visual Studio ist die Versionsverwaltung zum Glück direkt integriert. Visual Studio nutzt dafür die Quellcodeverwaltung Git. Welche Einsatzmöglichkeiten es gibt und wie Sie diese für Ihre eigenen Anforderungen einsetzen können, zeigt der vorliegende Artikel.

Das die Quellcodeverwaltung **Git** in Visual Studio integriert ist, sieht man beispielsweise an der Version 2019 direkt in der Menüleiste: Hier finden Sie den Eintrag **Git** mit einigen Unterpunkten vor (siehe Bild 1). In den folgenden Abschnitten erläutern wir allerdings erst einmal, in welchen Konstellationen Sie Git für Ihre Zwecke nutzen können.

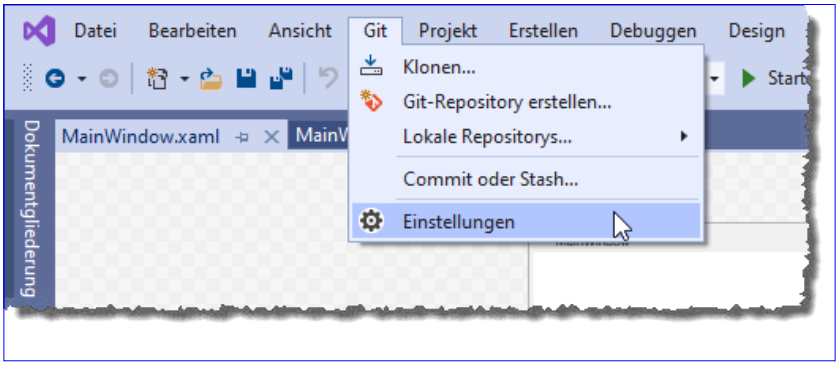


Bild 1: Git im Menüsystem von Visual Studio

Dabei verwenden wir die Begriffe **Quellcodeverwaltung** und **Versionsverwaltung** übrigens synonym.

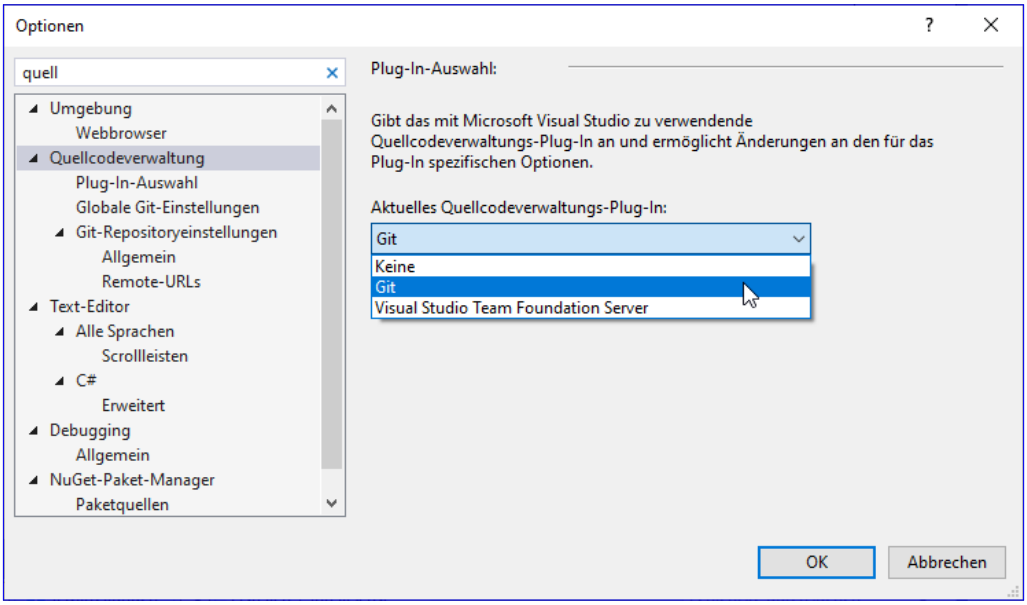


Bild 2: Quellcodeverwaltungs-Optionen

Quellcodeverwaltung festlegen

In den Optionen von Visual Studio können Sie unter Quellcodeverwaltung festlegen, welches Tool zur Quellcodeverwaltung eingesetzt werden soll (siehe Bild 2).

Unter Visual Studio 2019 ist auf unserem Testsystem **Git** bereits voreingestellt.

Einsatzzwecke der Versionsverwaltung mit Git

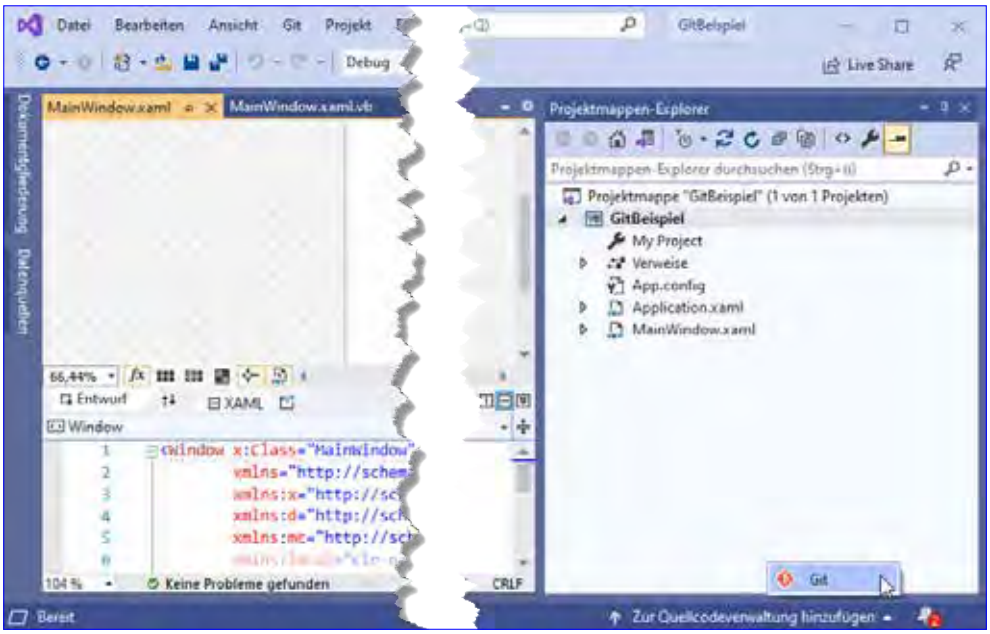


Bild 3: Hinzufügen zu Git

Git wurde 2005 vom Erfinder von Linux, Linus Thorvalds, entwickelt. Die eigentliche Idee war damals, dass man die unterschiedlichen Versionen einer Software auf einem Server speichert, damit alle an dem Projekt beteiligten Entwickler darauf zugreifen konnten. Das unter anderem zu dem Zweck, jederzeit eine aktuelle Version des Projekts herunterladen zu können und somit immer auch die Änderungen der anderen Entwickler präsent zu haben, aber auch dazu, die eigenen Änderungen mit den anderen Entwicklern zu teilen. Gegebenenfalls folgt hier noch ein Zwischenschritt, in dem die Änderungen noch geprüft werden, bevor sie in die Hauptversion übernommen werden.

Sie können Git aber auch einfach nutzen, um die verschiedenen Versionsstände Ihrer eigenen Projekte auf dem lokalen Rechner zu verwalten. Weiter oben haben wir bereits das Beispiel von Access genannt: Hier konnte man eigentlich nur die komplette Datenbank sichern, um gegebenenfalls zu einem bestimmten Versionsstand zurückspringen zu können. Bei Visual Studio-Produkten ist es selbst ohne Quellcodeverwaltung bereits möglich, einfach den Ordner mit dem Projekt zu kopieren und sich nach Bedarf eine ältere Version einer oder mehrerer Dateien in den aktuellen Projektordner zu holen. Mit Git wird das alles noch viel einfacher – wie das gelingt, zeigen die folgenden Abschnitte.

Projekt zu Git hinzufügen

Wenn Sie ein bestehendes Projekt haben und dieses mit der Quellcodeverwaltung **Git** verwalten möchten, finden Sie den notwendigen Befehl direkt unten rechts im Hauptfenster von Visual Studio (siehe Bild 3). Klicken Sie auf **Zur Quellcodeverwaltung hinzufügen**, erscheint eine Liste der verfügbaren Quellcodeverwaltungen – im Screenshot also nur **Git**.

Wählen Sie diesen Eintrag aus, erscheint als Nächstes der Dialog **GIT-Repository erstellen** aus Bild 4.

Hier wollen wir fürs Erste nur ein lokales Repository anlegen. Daher aktivieren wir links den Eintrag **Nur lokal**. Dies vereinfacht den Dialog stark, sodass wir nur noch den Pfad für das lokale Git-Repository angeben müssen. Hier wollen wir einfach das Hauptverzeichnis des Projekts nutzen.

Ein Klick auf die Schaltfläche **Erstellen** legt das Repository an und fügt bereits alle Dateien des Projekts hinzu. Der Projektmappen-Explorer präsentiert sich nun leicht verändert und auch in dem Bereich des Hauptfensters von Visual Studio, wo sich zuvor der Befehl **Zur Quellcodeverwaltung hinzufügen** befunden hat, zeigen sich einige neue Elemente (siehe Bild 5).

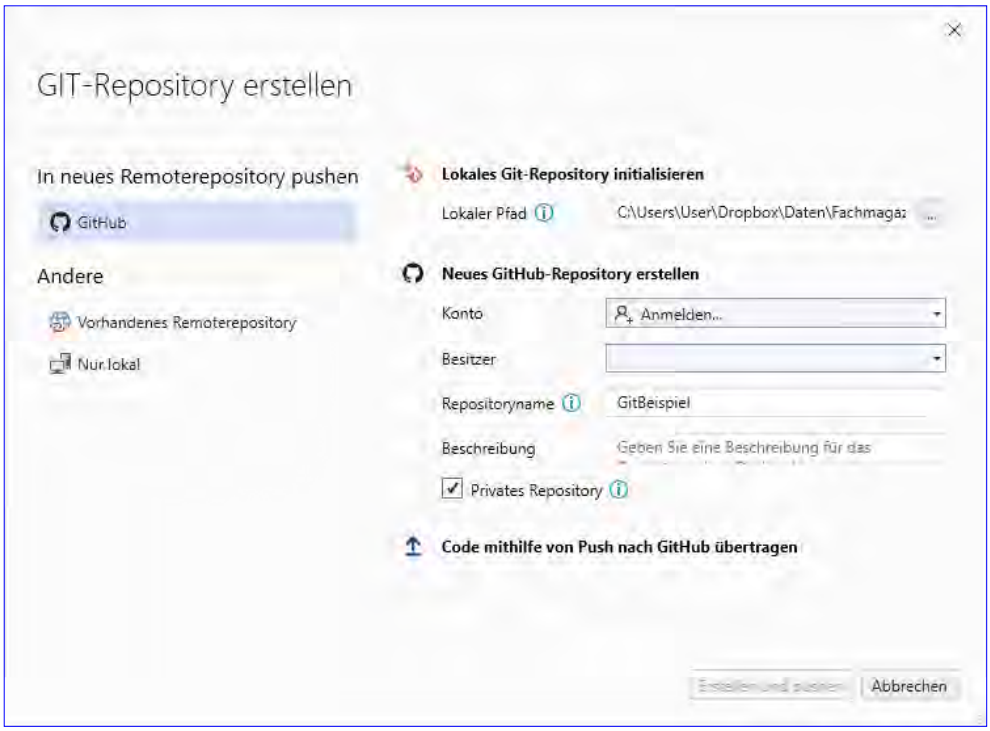


Bild 4: Erstellen eines Git-Repositorys

Hier sehen Sie zum Beispiel, wie das Repository heißt und dass Sie sich aktuell in einem sogenannten Branch namens **master** befinden (mehr dazu weiter unten). Klicken Sie auf master und zeigen für den nun erscheinenden Eintrag master das Kontextmenü an, finden Sie unter anderem einen Befehl namens Verlauf anzeigen (siehe Bild 6). Mit einem Klick auf diesen Befehl zeigen Sie alle bisherigen Versionen des ganzen Projekts an. Später werden Sie sehen, dass Sie auch die Versionen für einzelne Dateien

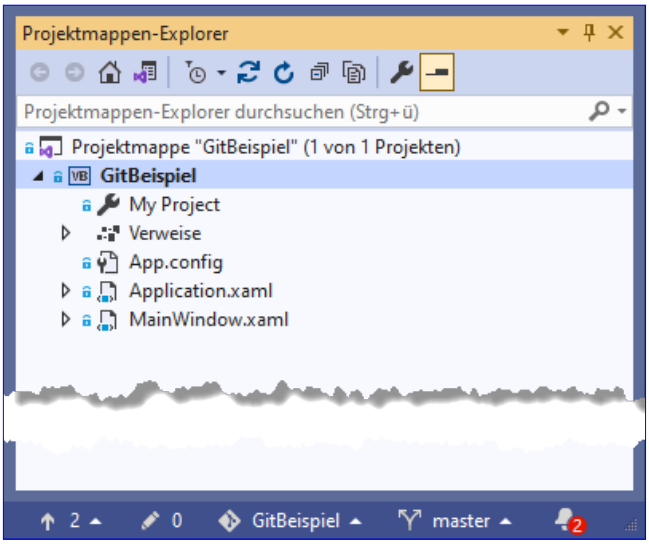


Bild 5: Der veränderte Projektmappen-Explorer

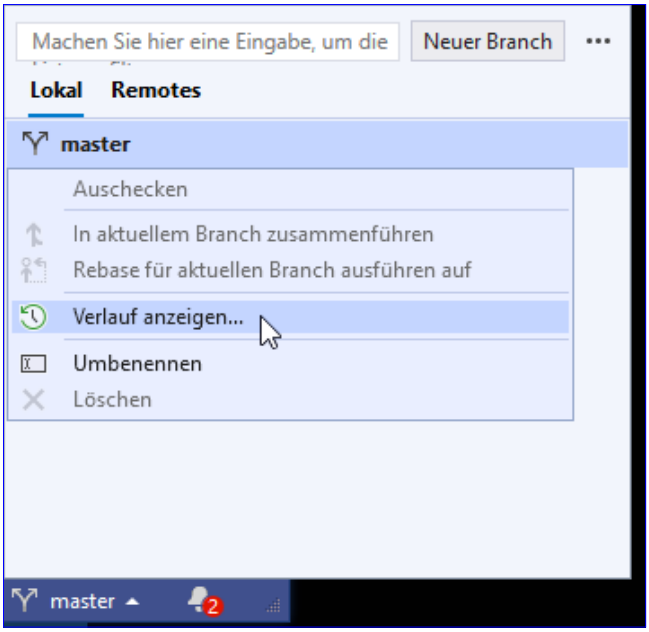


Bild 6: Anzeigen des Verlaufs

anzeigen können. An diesem Beispiel zeigen wir dann auch die Liste der bisher erstellten Versionen.

Nun zu den Einträgen der Projektmappe: Hier finden wir jeweils ein Schloss-Symbol vor, das beim Überfahren mit der Maus den Text **Eingecheckt** anzeigt. Ist dieses Symbol vorhanden, sehen Sie im Projektmappen-Explorer die Version des Elements, wie es auch im Repository eingecheckt ist. Eingecheckt heißt in diesem Zusammenhang, dass die aktuelle Version dort gespeichert ist.

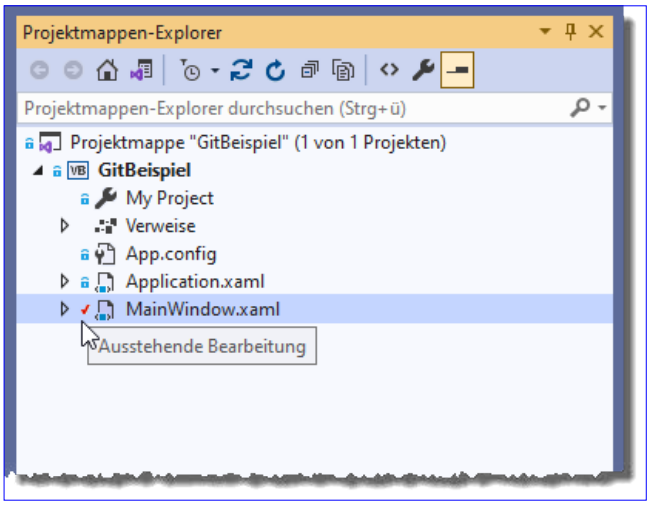


Bild 7: Ein Element mit ausstehender Bearbeitung

Eine Änderung des Symbols eines Elements erhalten wir beispielsweise, wenn wir seinen Code ändern. Dann wandelt Visual Studio das Symbol in ein rotes Häkchen um und der angezeigte Text lautet **Ausstehende Bearbeitung** (siehe Bild 7).

Wenn Sie die neue Version nun einchecken wollen, wählen Sie den Kontextmenü-Eintrag **Git|Commit oder Stash** für dieses Element aus (siehe Bild 8).

Dies ruft den Bereich aus Bild 10 hervor. Hier geben Sie in das Textfeld mit dem Text **Nachricht eingeben** einen Text ein, der die aktuellen Änderungen beschreibt. Auf diese Weise können Sie später nachvollziehen, welche Änderungen dieses Commit enthält. Klicken Sie dann auf **Commit für alle**, wird der Bereich geschlossen und das geänderte Element erhält im Projektmappen-Explorer das Icon **Eingecheckt**.

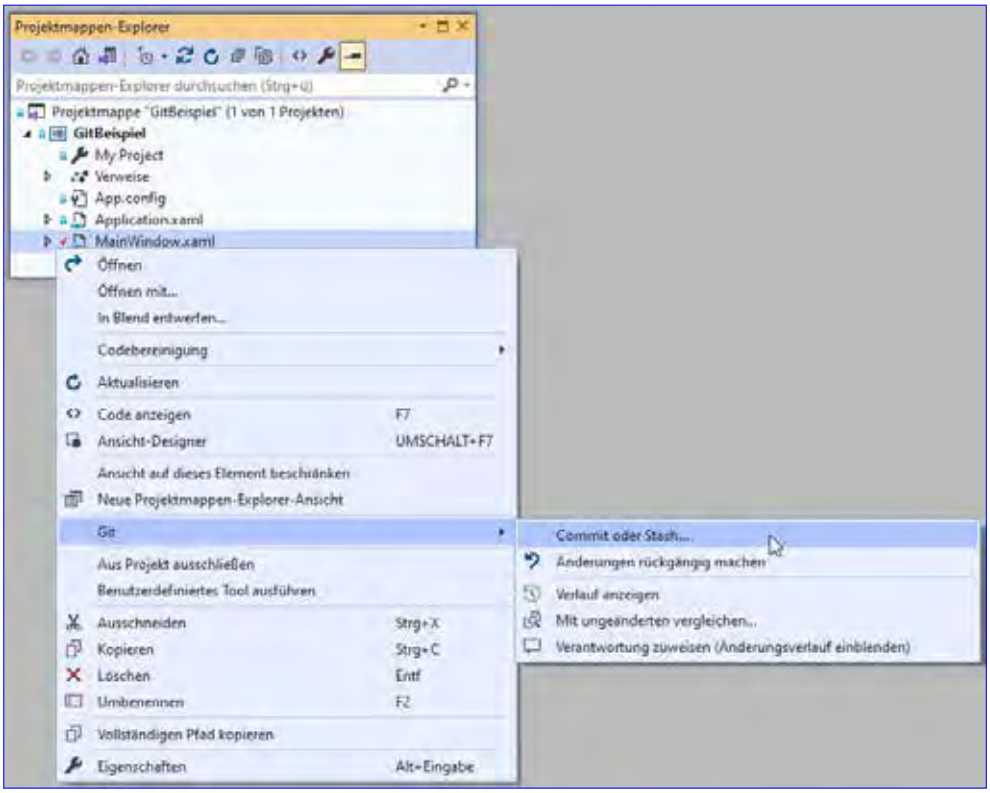


Bild 8: Element einchecken

Was ist passiert?

Durch das Einchecken oder Commit einer oder

mehrerer Dateien mit Änderungen sorgen Sie für die folgenden Änderungen:

- Die Datei wieder als **Eingecheckt** markiert.
- Im Repository sind nun zwei Versionsstände dieser Datei hinterlegt.
- Es liegen einige neue Elemente im Dateisystem vor (siehe Bild 9).

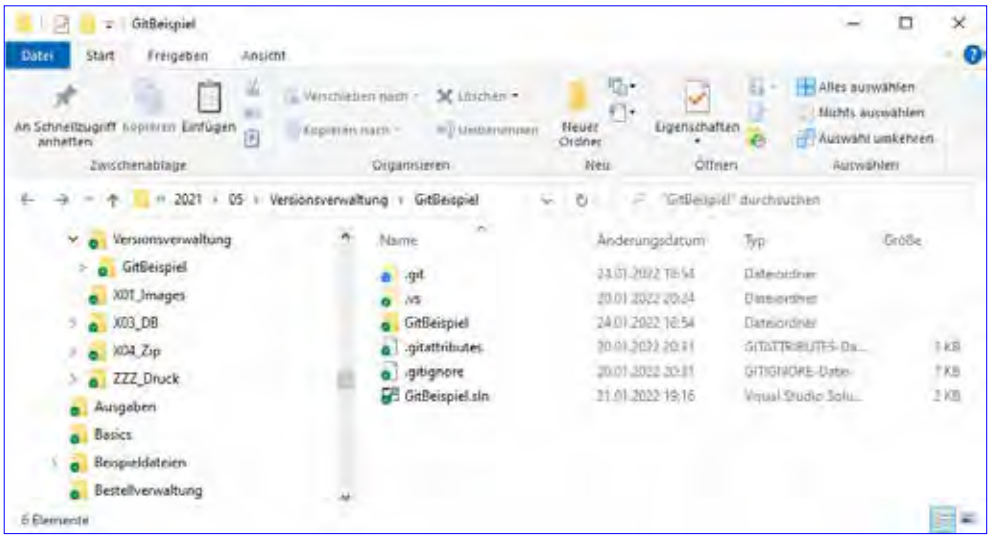


Bild 9: Elemente des lokalen Repositorys

Die Datei **.gitignore** enthält beispielsweise die Namen aller Dateien, die von der Quellcodeverwaltung ignoriert werden sollen.

Hierzu gehören beispielsweise die beim Debuggen erstellten Dateien oder auch benutzerspezifische Files.

Und Sie können nun verschiedene Aktionen anstoßen:

- Anzeigen des Verlaufs
- Anzeigen der Unterschiede zwischen den verschiedenen Versionen
- Wiederherstellen älterer Versionen

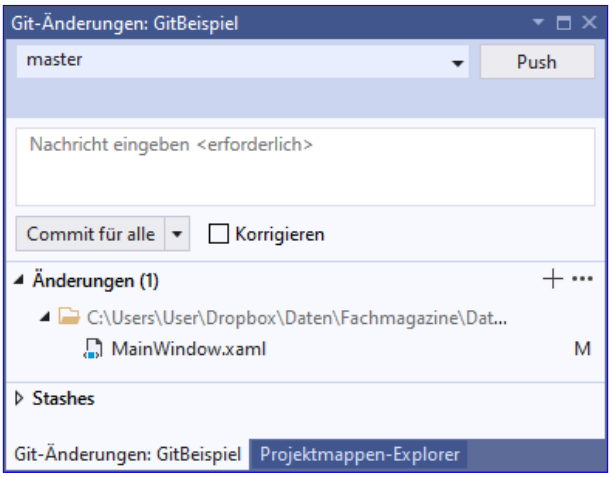


Bild 10: Einstellungen für das Commit

Verlauf ansehen

Der Verlauf listet alle Versionen einer Datei auf.

Um diese anzuzeigen, klicken Sie mit der rechten Maustaste auf die Datei im Projektmappen-Explorer, die Sie untersuchen wollen, und wählen dort

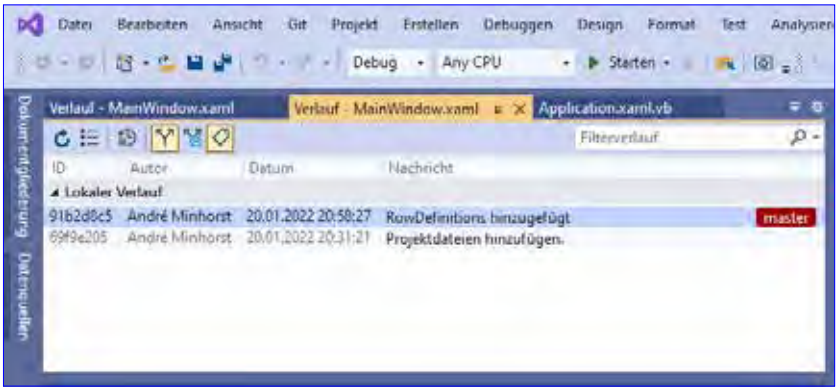


Bild 11: Anzeige des bisherigen Verlaufs

Beispieldaten generieren mit Bogus

Wenn Sie eine Anwendung entwickeln, können Sie die meisten Funktionen nur testen, wenn die zugrunde liegende Datenbank bereits Beispieldaten enthält. Ausnahmen sind beispielsweise Formulare zum Anlegen neuer Datensätze – hier legen Sie die Beispieldatenbank selbst an. In allen anderen Fällen kann es nicht schaden, ein paar Beispieldatensätze in den jeweiligen Tabellen bereitzustellen. Oft reicht es aus, das Anlegen einiger Elemente wie Kunden oder Produkte hart im Code zu verdrahten. Wenn Sie allerdings flexibel schnell für verschiedene Datenmodelle und Anwendungen Beispieldaten benötigen, reduzieren Sie den Aufwand zum Zusammenstellen der Beispieldaten zwar auf das einmalige Schreiben des Codes. Es macht aber auch keinen Spaß, sich dann Informationen wie Teststraße, Beispielfirma und Co. aus den Fingern zu saugen. Um dies zu automatisieren und gegebenenfalls auch größere Mengen an Beispieldaten zu generieren, gibt es spezielle Tools. Eines davon heißt Bogus – und diese stellen wir in diesem Artikel im Detail vor.

Für Menschen wie mich, die immer wieder frische Daten für die Beispiele für die Artikel und Bücher benötigen, ist ein Tool zum automatischen Generieren von Beispielcode natürlich Gold wert. Aber auch Sie können sich vermutlich eine Menge Arbeit sparen, wenn Sie die Tabellen Ihrer Anwendung vor dem Testen schnell mit passenden Beispieldaten füllen können.

Wie für die meisten Anwendungsfälle, die Sie mit Visual Studio abarbeiten, finden sich auch hier passende Erweiterungen im NuGet-Paket-Manager. Das Paket, das wir in diesem Artikel vorstellen wollen, heißt Bogus und bietet bereits in der kostenlosen Variante sehr viele verschiedene mögliche Generatoren für Daten unterschiedlichster Art an.

Um zuvor einmal die grundlegende Idee zu skizzieren: Wir gehen davon aus, dass Sie in der zu programmierenden Anwendung beispielsweise eine Tabelle namens **Kunden** verwenden, dessen Einträge im Entity Data Model der Anwendung jeweils in einem Element namens **Kunde** gespeichert werden. Das Tool, in diesem Fall Bogus, soll uns eine Möglichkeit bieten, die Eigenschaften neu erstellter Elemente des Typs **Kunde** mit Werten zu füllen.

Wenn Kunde also beispielsweise Eigenschaften wie **Anrede**, **Vorname**, **Nachname**, **Strasse**, **PLZ** und **Ort** enthält, dann müsste das Tool Funktionen bereitstellen, mit denen Werte für genau diese Felder erzeugt werden können und die nach dem Zufallsprinzip ausreichend neue Werte liefern.

Noch spannender wird es an der Stelle, wo wir verknüpfte Tabellen haben – zum Beispiel **Kunden** und **Bestellungen**. Im einem Entity Data Model haben wir dann zwei Klassen namens **Kunde** und **Bestellung**, deren Eigenschaften wir nacheinander füllen müssen. Allerdings müssen wir zuerst die Kunden erstellen und wenn wir in einer Bestellung den Kunden referenzieren, der diese Bestellung aufgegeben hat, müssen wir auf einen der zuvor erstellten Kunden verweisen. Auch diese Möglichkeit bietet das hier verwendete Tool **Bogus**.

Beispielanwendung

Um die Funktionen von Bogus schnell und komfortabel testen zu können, nutzen wir **LINQPad5**. Hier können wir Code schnell eingeben und ausführen, ohne immer direkt ein Projekt erstellen zu müssen. Wenn Sie **LINQPad5** geöffnet haben, sind zwei Schritte nötig: der Wechsel zu einem passenden Query-Typ, hier **VB Program**, und das Hinzufügen von Bogus zur Query.

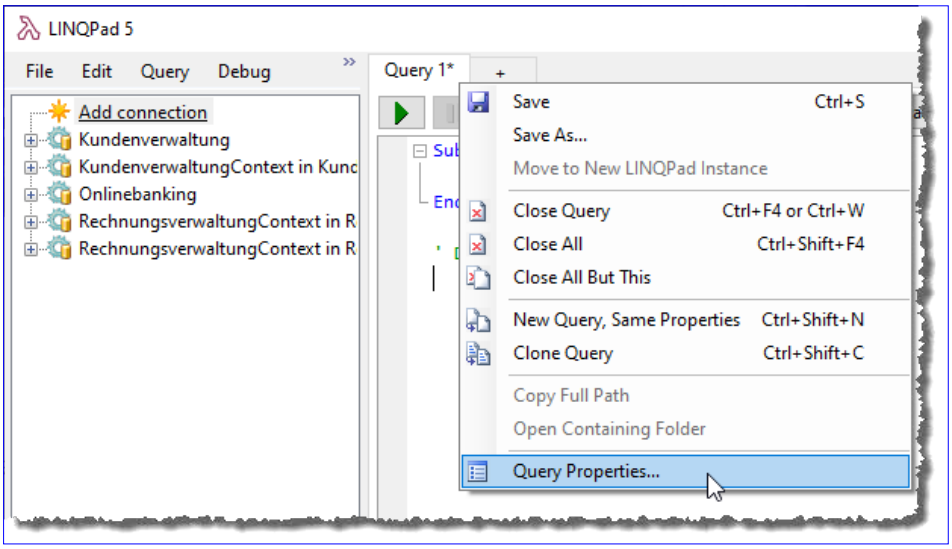


Bild 1: Aufrufen der Query-Eigenschaften

Bogus zur Query hinzufügen

Um das Bogus-Paket zur Query hinzuzufügen, klicken Sie mit der rechten Maustaste auf den Registerreiter für die Query. Dort wählen Sie den Eintrag **Query Properties...** aus (siehe Bild 1).

Dies öffnet den Dialog **Query Properties**. Hier klicken Sie unten auf die Schaltfläche **Add NuGet...**, was den Dialog **LINQPad NuGet Manager** öffnet. Hier geben Sie im mittleren Bereich den Suchbegriff **Bogus** ein und klicken dann für den obersten Eintrag auf die Schaltfläche **Add to Query** (siehe Bild 2). Schließen Sie den Dialog, finden Sie den Eintrag **Bogus** nun im Dialog **Query Properties** im Bereich **Additional Reference** vor.

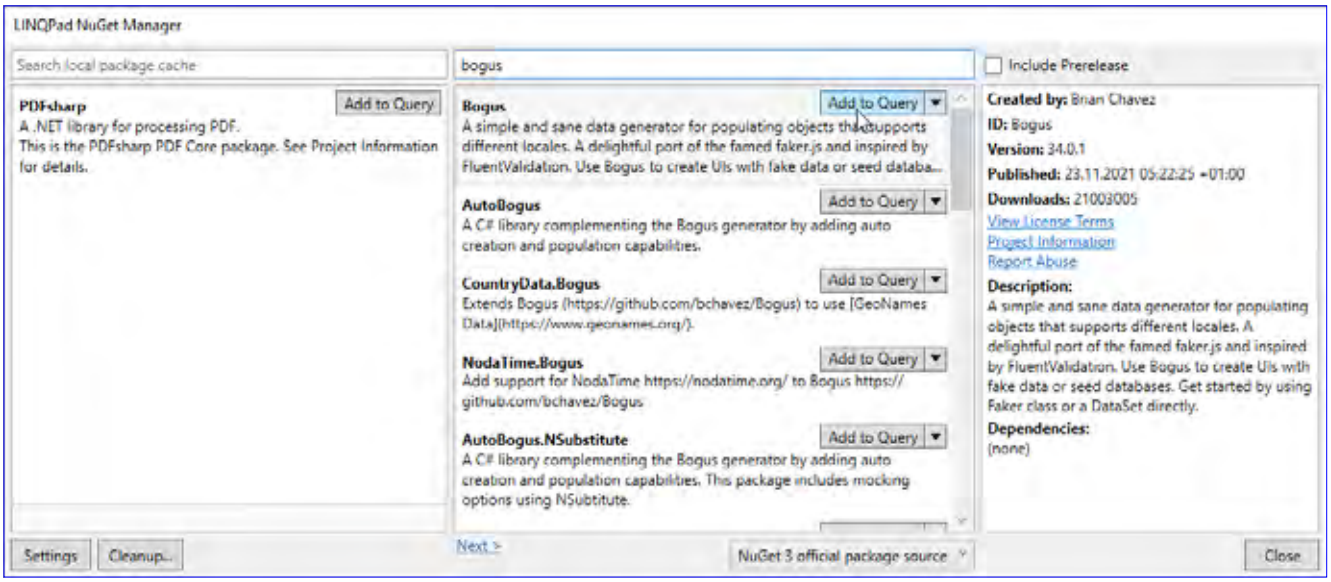


Bild 2: Hinzufügen des Pakets **Bogus**

Beispielkunde erstellen

Der Rahmen für die ersten Experimente mit Bogus ist eine einfache Kundenklasse. Diese definieren Sie wie folgt unterhalb der in der LINQPad-Query vorgegebenen Methode **Main**:

```
Public Class Kunde
    Property Vorname As String
    Property Nachname As String
    Property Strasse As String
    Property PLZ As String
    Property Ort As String
    Property Land As String
    Property EMail As String
    Property Telefon As String
End Class
```

In der Methode **Main** werden wir gleich die Beispielmethoden aufrufen. Die erste heißt **KundeAnlegen**:

```
Sub Main
    KundeAnlegen
End Sub
```

Bogus nutzen

Die einfachste Möglichkeit, Bogus zu verwenden, ist das Initialisieren einer Objektvariablen auf Basis des Objekts **Bogus.Faker** und das anschließende Erzeugen der verschiedenen Zufallsdaten.

Das folgende Beispiel erledigt das und gibt so beispielsweise ein zufällig ermitteltes Land aus:

```
Public Sub AndereBeispiele
    Dim objFaker As New Bogus.Faker()
    Debug.Print (objFaker.Address.Country)
End Sub
```

Auf Basis dieses Beispiels könnten Sie im Grunde schon die Eigenschaften von neu erzeugten Objekten füllen. Es gibt jedoch noch praktischere Methoden. Mit der soeben vorgestellten Methode können Sie sich allerdings schon einmal durch die einzelnen Klassen und deren Eigenschaften arbeiten und sich die Möglichkeiten anschauen. Weiter unten finden Sie eine Beschreibung der einzelnen Klassen.

Objekte mit Bogus füllen

Nun schauen wir uns in der Methode **KundeAnlegen** am Beispiel eines neuen Kunden an, wie Sie Bogus nutzen können. Dabei erstellen wir zuerst ein neues **Kunde**-Objekt und ein neues Objekt des Typs **Bogus.Faker** für unsere Klasse **Kunde**:

```
Public Sub KundeAnlegen
    Dim Beispielkunde As New Kunde
    Dim objFaker As New Bogus.Faker(Of Kunde)
```

Das **Faker**-Objekt ist unser eigentlicher Generator. Für diesen legen wir über die **RuleFor**-Eigenschaft fest, welche Eigenschaft des Zielobjekts mit welcher Eigenschaft des Fakers gefüllt wird. Die **RuleFor**-Definitionen legen wir einfach in einer langen Zeile jeweils durch einen Punkt getrennt hintereinander an – der Übersichtlichkeit halber haben wir hier Zeilenumbrüche eingefügt. Die erste Regel besagt beispielsweise, dass wir die Eigenschaft **Vorname** mit der Eigenschaft **Name.FirstName** des Fakers füllen wollen, die weiteren Regeln legen die Quelle für die übrigen Eigenschaften fest:

```
    objFaker.RuleFor(Function(k) k.Vorname, Function(f) f.Name.FirstName) _
        .RuleFor(Function(k) k.Nachname, Function(f) f.Name.LastName) _
        .RuleFor(Function(k) k.Strasse, Function(f) f.Address.StreetAddress) _
        .RuleFor(Function(k) k.PLZ, Function(f) f.Address.ZipCode) _
        .RuleFor(Function(k) k.Ort, Function(f) f.Address.City) _
        .RuleFor(Function(k) k.Land, Function(f) f.Address.Country) _
        .RuleFor(Function(k) k.EMail, Function(f) f.Internet.Email) _
        .RuleFor(Function(k) k.Telefon, Function(f) f.Phone.PhoneNumber)
```

Dabei nutzen wir nicht nur die **Name**-Klasse (für **FirstName** und **LastName**), sondern auch die **Address**-Klasse (für alle Adress-relevanten Daten) sowie die **Internet**-Klasse (für die E-Mail-Adresse aus der Eigenschaft **Email**) und die **Phone**-Klasse (für die Telefonnummer aus **PhoneNumber**). Danach folgt das eigentliche Erstellen. Hier rufen wir die **Generate**-Methode des **Faker**-Objekts auf und schreiben das Ergebnis vom Typ **Kunde** in die Variable **Beispielkunde**:

```
    Beispielkunde = objFaker.Generate
```

Die Werte der Eigenschaften des neuen **Kunde**-Objekts schreiben wir danach wie folgt in den Debug-Bereich:

```
    Debug.Print("Vorname: " & Beispielkunde.Vorname)
    Debug.Print("Nachname: " & Beispielkunde.Nachname)
    Debug.Print("Strasse: " & Beispielkunde.Strasse)
    Debug.Print("PLZ: " & Beispielkunde.PLZ)
    Debug.Print("Ort: " & Beispielkunde.Ort)
    Debug.Print("Land: " & Beispielkunde.Land)
    Debug.Print("E-Mail: " & Beispielkunde.EMail)
    Debug.Print("Telefon: " & Beispielkunde.Telefon
End Sub
```

Das Ergebnis nach dem ersten Aufruf sieht beispielsweise wie folgt aus:

```
Vorname: Annamae
Nachname: Kling
Strasse: 02613 Schmidt Light
PLZ: 51900
Ort: Rachelleshire
Land: Ecuador
E-Mail: Barton_Casper58@yahoo.com
Telefon: (985) 647-9446
```

Einfachere Schreibweise

Wenn Sie die Schreibweise oben mit den verschachtelten **Function**-Anweisungen zu kompliziert finden, können Sie auch eine einfachere Schreibweise nutzen. Dazu verwenden wir eine einzige **Function**-Anweisung, die **f** und **k** gleichzeitig als Parameter nutzt. Dadurch können Sie innerhalb der **Function** nun einfache Zuweisungen nutzen wie beispielsweise **k.Vorname = f.Name.FirstName**. Wichtig ist, dass Sie die Parameter in der Function in umgekehrter Reihenfolge wie bei den anschließenden Zuweisungen angeben. Die folgende Version zum Anlegen eines Kunden funktioniert genauso wie die vorherige, lässt sich aber viel leichter lesen und auch bearbeiten:

```
Public Sub KundeAnlegenKompakt
    Dim Beispielkunde As New Kunde
    Dim objFaker As New Bogus.Faker(Of Kunde)
    objFaker.Rules(Function(f, k)
        k.Vorname = f.Name.FirstName
        k.Nachname = f.Name.LastName
        k.Strasse = f.Address.StreetAddress
        k.PLZ = f.Address.ZipCode
        k.Ort = f.Address.City
        k.Land = f.Address.Country
        k.EMail = f.Internet.Email
        k.Telefon = f.Phone.PhoneNumber
    End Function)
    Beispielkunde = objFaker.Generate
    ...
End Sub
```

Mehrere Objekte gleichzeitig erstellen

Mit der oben verwendeten **Generate**-Methode haben wir zunächst nur ein einziges **Kunde**-Objekt erstellt. Wenn Sie mehrere **Kunde**-Objekte gleichzeitig erstellen wollen, können Sie dies innerhalb einer Schleife mit der gewünschten Anzahl an Durchläufen erledigen. Es geht jedoch auch noch eleganter. Dazu legen wir in einer weiteren Beispielmethode eine **List**-Variable namens **Kunden** an, die wir für die Aufnahme von **Kunde**-Elemente auslegen. Dann legen wir die üblichen Regeln für das Anlegen der **Kunde**-Elemente fest. Anschließend rufen wir wieder die **Generate**-Methode des **Faker**-Objekts auf, diesmal jedoch übergeben wir als Parameter die Anzahl

Beispieldaten für ein EDM generieren

Im Artikel »Beispieldaten generieren mit Bogus« haben wir gezeigt, wie Sie grundsätzlich Beispieldaten mit der Erweiterung Bogus erzeugen. Dort haben wir allerdings noch offen gelassen, wie Sie solche Daten erzeugen, die in verknüpften Tabellen gespeichert werden sollen – also beispielsweise in zwei Tabellen namens »Kunden« und »Bestellungen«, wobei die Tabelle »Bestellungen« über ein Fremdschlüsselfeld namens »KundeID« mit der Tabelle »Kunden« verknüpft ist. Wie das gelingt, und welche Techniken noch interessant sind für das Schreiben von Beispieldaten über ein Entity Data Model direkt in die zugrunde liegenden Tabellen, beschreiben wir im vorliegenden Artikel.

Beispielprojekt erstellen

Wir erstellen ein einfaches Beispielprojekt des Typs **WPF-App** (für **Visual Basic**) namens **BeispieldatenGenerieren**. Diesem fügen wir ein Entity Data Model hinzu, das wir **BeispieldatenContext** nennen (Menüeintrag **Projekt|Neues Element hinzufügen...**) und das den Typ **Leeres Code First-Modell** erhalten soll. Für dieses Projekt legen wir als Erstes eine Klasse namens **Kunde** an, die wie folgt aussieht:

```
Partial Public Class Kunde
    Public Overridable Property ID As Int32
    Public Overridable Property Vorname As String
    Public Overridable Property Nachname As String
    Public Overridable Property Firma As String
    Public Overridable Property Strasse As String
    Public Overridable Property PLZ As String
    Public Overridable Property Ort As String
    Public Overridable Property Land As String
    Public Overridable Property EMail As String
    Public Overridable Property Bestellungen As ICollection(Of Bestellung)
End Class
```

Außerdem fügen wir eine Klasse namens **Bestellung** hinzu:

```
Partial Public Class Bestellung
    Public Overridable Property ID As Int32
    Public Overridable Property Bestelldatum As DateTime
    Public Overridable Property KundeID As Int32
    Public Overridable Property Kunde As Kunde
    Public Overridable Property Bestellpositionen As ICollection(Of Bestellposition)
End Class
```

Damit wir auch eine m:n-Beziehung abbilden können, legen wir noch eine Klasse **Produkt** und eine Klasse **Bestellposition** an:

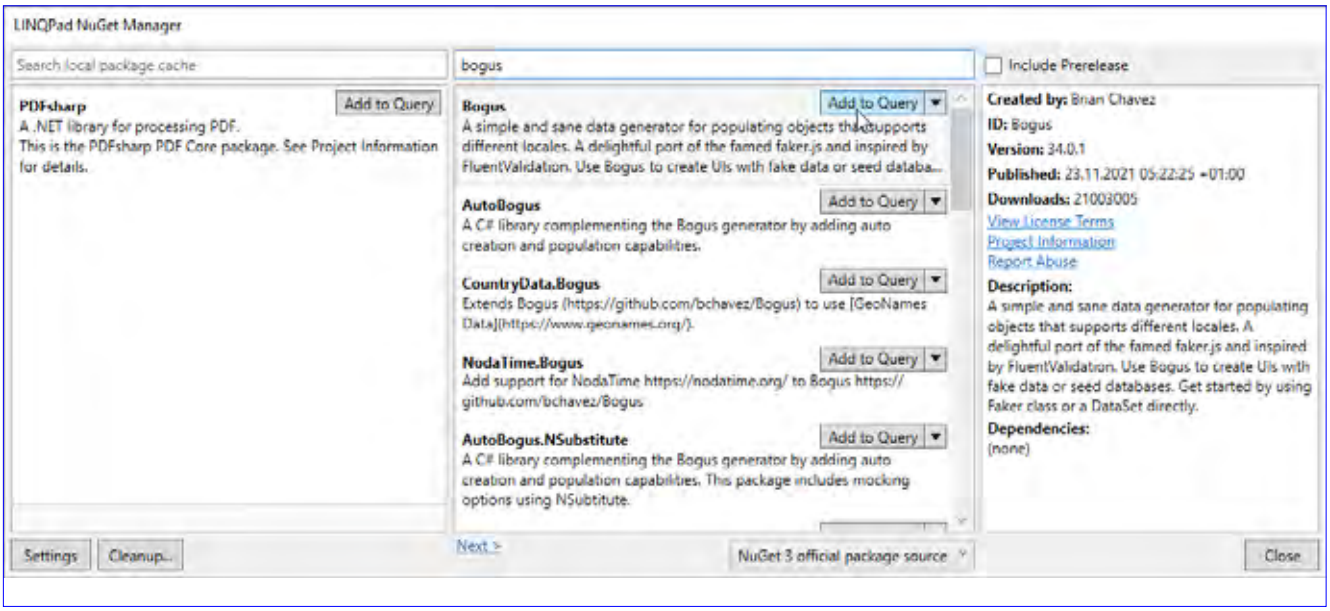


Bild 1: Hinzufügen des Pakets **Bogus**

```
Partial Public Class Bestellposition
    Public Overridable Property ID As Int32
    Public Overridable Property ProduktID As Int32
    Public Overridable Property BestellungID As Int32
    Public Overridable Property Produkt As Produkt
    Public Overridable Property Bestellung As Bestellung
    Public Overridable Property Einzelpreis As Decimal
    Public Overridable Property Menge As Int32
End Class

Partial Public Class Produkt
    Public Overridable Property ID As Int32
    Public Overridable Property Produktname As String
    Public Overridable Property Einzelpreis As Decimal
    Public Overridable Property Bestellpositionen As ICollection(Of Bestellposition)
End Class
```

Der Klasse **BeispieldatenContext.vb** fügen wir eine **DbSet**-Definition für die vorgestellten Klassen hinzu. Außerdem legen wir in der Konstruktor-Methode **New** fest, dass die noch zu definierende Klasse **BeispieldatenInitializer** zum Initialisieren der Datenbank verwendet werden soll:

```
Public Class BeispieldatenContext
    Inherits DbContext
    Public Sub New()
```

```
        MyBase.New("name=BeispieldatenContext")
        Database.SetInitializer(New BeispieldatenInitializer())
    End Sub
    Public Overridable Property Kunden() As DbSet(Of Kunde)
    Public Overridable Property Bestellungen() As DbSet(Of Bestellung)
    Public Overridable Property Produkte() As DbSet(Of Produkt)
    Public Overridable Property Bestellpositionen() As DbSet(Of Bestellposition)
End Class
```

Schließlich legen wir hier noch fest, dass die Elemente der Klassen **Kunde**, **Bestellung**, **Bestellposition** und **Produkt** in den Tabellen **Kunden**, **Bestellungen**, **Bestellpositionen** und **Produkte** landen sollen:

```
Protected Overrides Sub OnModelCreating(modelBuilder As DbModelBuilder)
    MyBase.OnModelCreating(modelBuilder)
    modelBuilder.Entity(Of Kunde)().ToTable("Kunden")
    modelBuilder.Entity(Of Bestellung)().ToTable("Bestellungen")
    modelBuilder.Entity(Of Bestellposition)().ToTable("Bestellpositionen")
    modelBuilder.Entity(Of Produkt)().ToTable("Produkte")
End Sub
```

Bogus zum Projekt hinzufügen

Danach fügen wir das Bogus-Paket zum Projekt hinzu. Dazu wählen Sie den Menüeintrag **Projekt|NuGet-Pakete verwalten** aus. Dies öffnet ein neues Fenster, indem Sie zum Bereich **Durchsuchen** wechseln. Hier geben Sie im Suchfeld den Text **Bogus** ein und finden gleich das passende Paket von **Brian Chavez** vor.

Markieren Sie dieses und starten die Installation mit einem Klick auf die Schaltfläche **Installieren** (siehe Bild 1).

Kundentabelle mit Bogus füllen

Danach starten wir direkt mit der Anwendung. Wir wollen erst einmal nur Daten in der Tabelle **Kunden** anlegen. Dazu fügen wir der Datenbank-Initialisierer-Klasse **Beispieldateninitialisierer** wie folgt eine **Seed**-Methode hinzu:

```
Public Class BeispieldatenInitializer
    Inherits DropCreateDatabaseAlways(Of BeispieldatenContext)

    Protected Overrides Sub Seed(context As BeispieldatenContext)
        Dim Kunden As List(Of Kunde) = New List(Of Kunde)
        Dim objFaker As Bogus.Faker(Of Kunde) = New Bogus.Faker(Of Kunde)("de")
        objFaker.Rules(Function(f, k) As Object
                           k.Vorname = f.Name.FirstName
                           k.Nachname = f.Name.LastName
```

```
                    k.Firma = f.Company.CompanyName
                    k.Strasse = f.Address.StreetAddress
                    k.PLZ = f.Address.ZipCode
                    k.Ort = f.Address.City
                    k.Land = "Deutschland"
                    k.EMail = f.Internet.Email
                End Function)
        Kunden = objFaker.Generate(1000)
        context.Kunden.AddRange(Kunden)
        MyBase.Seed(context)
    End Sub
End Class
```

Die **Seed**-Methode deklariert und initialisiert ein **List**-Objekt namens **Kunden** für Elemente des Typs **Kunde**. Dann erstellt sie ein sogenanntes **Faker**-Objekt namens **objFaker**, mit dem wir Elemente des Typs **Kunde** füllen wollen. Damit diese später beispielsweise deutsche Adressdaten verwendet und keine amerikanischen, hängen wir noch den Parameter **("de")** an.

Danach legen wir mit der Methode **Rules** die Regeln fest, nach denen neue Elemente mit Daten gefüllt werden sollen. Als Parameter für die **Rules**-Methode geben wir einen **Function**-Lambda-Ausdruck an. Dieser enthält die Zuweisungen der verschiedenen Klassen und Funktionen von Bogus (**f**) zu den Eigenschaften des zu erzeugenden Elements (**k**).

Die Eigenschaft **Vorname** füllen wir beispielsweise mit der Funktion **FirstName** der **Name**-Klasse. Wir möchten nur Kunden mit deutscher Adresse hinzufügen, also stellen wir für das Feld **Land** fix den Wert **Deutschland** ein. Nachdem wir die Zuweisung für alle Eigenschaften erstellt haben, rufen wir die **Generate**-Methode des Objekts aus **objFaker** auf und übergeben dieser als Parameter die Anzahl der zu erstellenden Elemente und weisen diese dem **List**-Objekt **Kunden** zu.

Diese Liste schreiben wir schließlich mit der **AddRange**-Methode in das **DbSet**-Element **Kunden**. Damit die Daten noch in die zugrunde liegende Tabelle der Datenbank geschrieben werden, rufen wir schließlich noch die **Seed**-Methode für den Datenbankkontext aus **context** auf.

Daten beim Start der Anwendung erstellen

Damit diese Daten beim Start der Anwendung geschrieben werden, benötigen wir einen Zugriff auf die Daten der Datenbank. Da wir eine WPF-Anwendung als Beispiel verwenden, die beim Start das Fenster **MainWindow.xaml** anzeigt, bietet es sich an, die notwendigen Anweisungen in die Konstruktor-Methode der zugrunde liegenden Klasse **MainWindow.xaml.vb** zu hinterlegen.

Diese sieht anschließend wie folgt aus und soll einfach die Kunden der Datenbank in die Liste **AlleKunden** einlesen:

```
Class MainWindow
    Public Sub New()
        Dim AlleKunden As List(Of Kunde)
        Dim dbContext As BeispieldatenContext
        dbContext = New BeispieldatenContext
        AlleKunden = New List(Of Kunde) _
            (dbContext.Kunden)
    End Sub
End Class
```

Starten Sie die Anwendung nun, stellt diese beim Zugriff auf den Datenbankkontext fest, dass die Datenbank noch nicht existiert und legt diese nach den Vorgaben in der Klasse **BeispieldatenInitializer** an.

Wenn Sie nun mit dem Menübefehl **Ansicht|SQL Server-Objekt-Explorer** den Bereich **SQL Server-Objekt-Explorer** anzeigen und dort zu der Datenbank navigieren, die in der Datei **App.config** im Bereich **connectionString** definiert ist, finden Sie hier bereits die frisch erstellten Tabellen vor (siehe Bild 2).

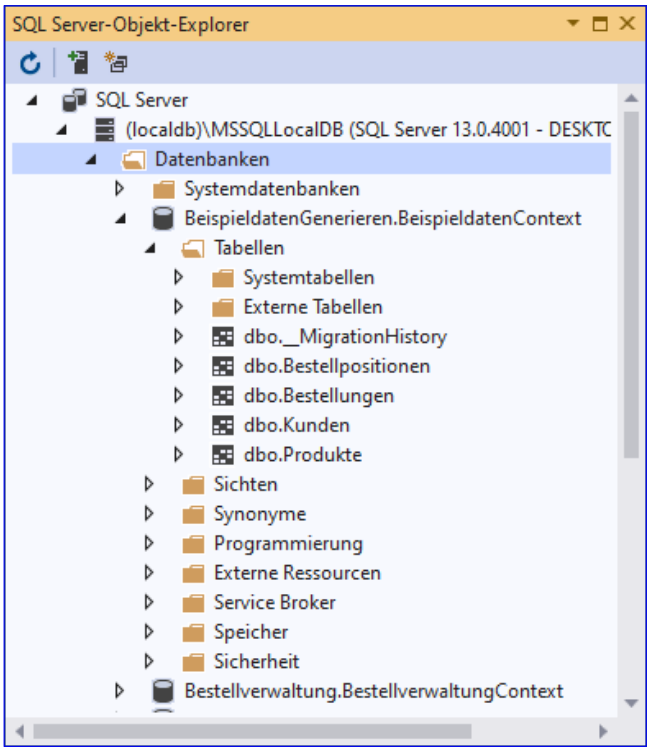


Bild 2: Die neu erstellte Datenbank samt Tabellen im SQL Server-Objekt-Explorer

Bild 3: Die Tabelle mit den Kundendaten

Seminarverwaltung I: Entity Data Model

Es wird Zeit, die gelernten Techniken mal wieder an einer praktischen Lösung auszuprobieren. In diesem Fall soll es eine Anwendung werden, die ich selbst einsetzen will, da ich seit ein paar Wochen auch Webinare zu verschiedenen Themen anbiete. Eine der Herausforderungen besteht darin, die Kunden, die über einen Onlineshop bestellt haben, in die Anwendung einzulesen und diese den entsprechenden Seminaren beziehungsweise Webinaren zuzuordnen. Zu gegebener Zeit sollen die Teilnehmer eine Mail mit dem Link zur Teams-Sitzung erhalten und anschließend noch einen Link mit dem Download der Aufzeichnung des Seminars. Schließlich sollen auch noch Zertifikate über die Teilnahme erstellt und versendet werden. Wie dies alles gelingt, zeigt der vorliegende Artikel.

Projekt anlegen

Die Gestaltung einer Anwendung zum Verwalten von Daten startet mit dem Anlegen eines Projekts. Wir verwenden ein Projekt des Typs **WPF-App (.NET Framework)** mit der Sprache Visual Basic und auf Basis von XAML für die Gestaltung der Benutzeroberfläche.

Entity Data Model hinzufügen

Wir wollen die Daten in einer SQL Server-Datenbank speichern und über ein Entity Data Model auf diese Daten zugreifen. Dazu fügen wir dem Projekt ein Entity Data Model hinzu. Mit **Strg + Umschalt + A** öffnen wir den Dialog **Neues Element hinzufügen** und wählen dort den Eintrag **ADO.NET Entity Data Model** aus. Nach der Angabe des Namens **SeminarverwaltungContext** klicken wir auf die Schaltfläche **Hinzufügen** (siehe Bild 1).

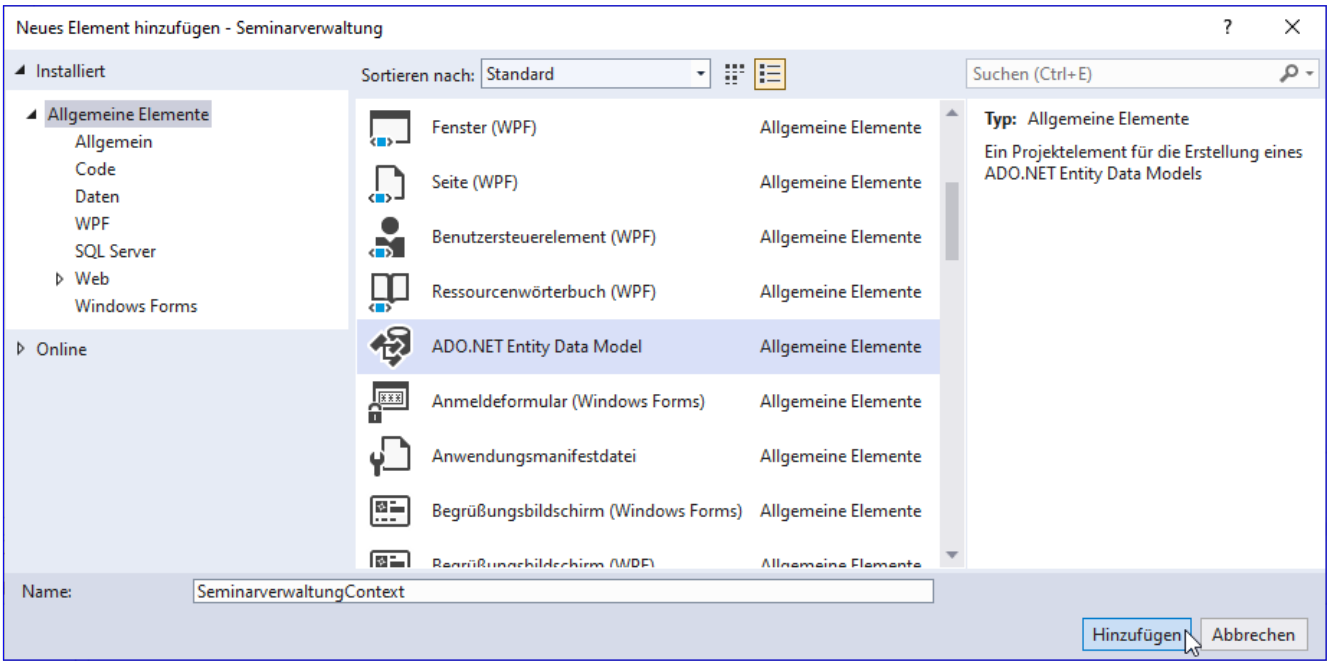


Bild 1: Hinzufügen eines Entity Data Models

Danach wählen Sie im Dialog **Assistent für Entity Data Model** den Eintrag **Leeres Code First-Modell** aus (siehe Bild 2).

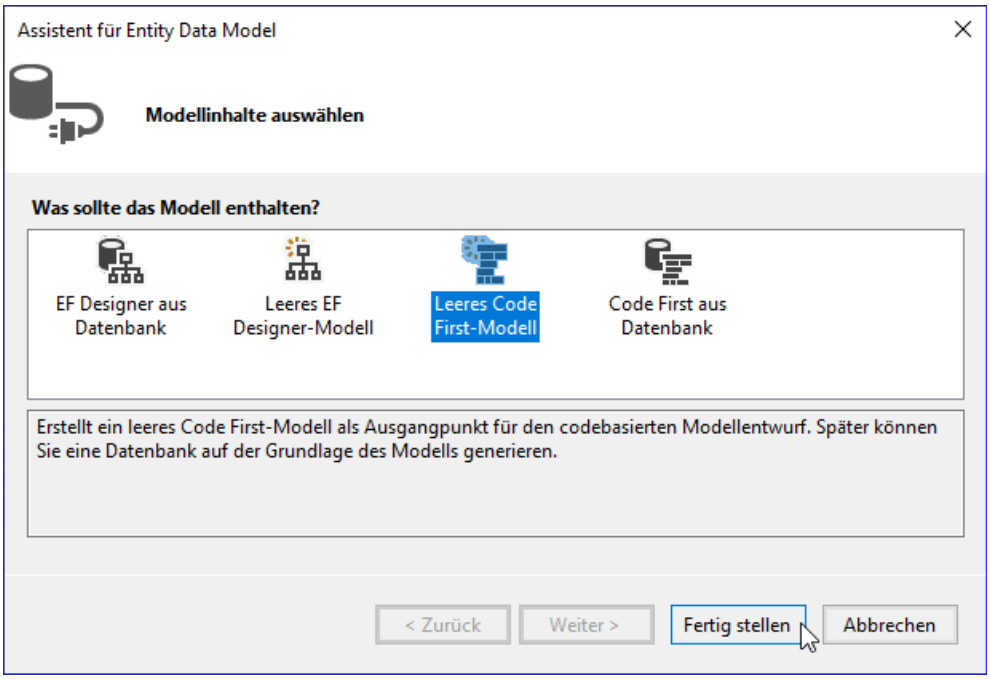


Bild 2: Auswahl des Model-Typs

Nach wenigen Sekunden erscheint die Klasse **Seminarverwaltung-Context**, in der wir für jede zu erstellende Entität beziehungsweise Tabelle einen Eintrag hinzufügen. Hier finden wir auch eine Vorlage für eine Entitätsklasse.

Entitäten hinzufügen

Damit können wir beginnen, die gewünschten Entitäten hinzuzufügen. Dazu überlegen wir uns zunächst, welche Entitäten wir benötigen. Wir benötigen auf jeden Fall eine Entität für die Kunden und eine für die Seminare. Jedem Kunden soll jedes Seminar zugeordnet werden können und umgekehrt. Dafür soll später in der Datenbank eine Verknüpfungstabelle angelegt werden. Damit dies geschieht, benötigen wir keine eigene Entität für die Verknüpfungstabelle – die Beziehungstabelle erstellen wir dann über ein spezielles Mapping.

Die Kunden haben eine Anrede, die wir in einer eigenen Tabelle speichern und dann über ein Fremdschlüsselfeld zuweisen wollen, also legen wir auch eine entsprechende Entität für die Anreden an.

Außerdem benötigen wir zumindest eine Tabelle zur Verwaltung der Seminare. An dieser Stelle kann man sich gleich überlegen, ob es nur eine Tabelle zum Speichern der Seminare gibt. Dann muss man Seminare mit Themen, die nochmal wiederholt werden sollen, immer wieder komplett neu anlegen. Man könnte auch eine Tabelle mit Seminaren erstellen, welche die Basisinformationen eines Seminars enthält wie den Titel, die Inhaltsangabe, den Preis et cetera – beispielsweise namens **Seminarthemen**.

Und dann fügt man eine neue Tabelle namens **Seminare** hinzu, welche die Seminartermine enthält und das Datum mit dem Seminar verknüpft. Oder man macht es gleich wie beim Beispiel der Bestellpositionen in einer Bestellverwaltung, wo man die Details des Artikels wie Einzelpreis oder Steuersatz aus der Artikeltabelle übernimmt, damit diese auch nach Änderung von Preis oder Steuersatz in der Artikeltabelle in der Tabelle für Bestellpositionen enthalten bleiben. In diesem Fall würden wir dann die Felder **Titel**, **Inhalt** und **Preis** von der Tabelle **Seminarthemen** in die Tabelle **Seminare** übernehmen.

Schließlich benötigen wir noch die Verknüpfungstabelle zwischen den Tabellen **Seminare** und **Kunden**, die wir, wie oben beschrieben, allein über Mapping mit der Fluent API definieren. Diese Tabelle soll **SeminareKunden** heißen.

Die Entität Kunde

Für die Entität **Kunde** legen wir die üblichen Eigenschaften wie Vorname, Nachname, Adresse et cetera fest. Wichtig ist hier vor allem die E-Mail-Adresse. Schließlich definieren wir hier eine **ICollection** namens **Seminare** für die Elemente des Typs **Seminar**. Diese wird im Konstruktor der Entitätsklasse erstellt:

```
Public Class Kunde
    Public Overridable Property ID As Int32
    Public Overridable Property Vorname As String
    Public Overridable Property Nachname As String
    Public Overridable Property Firma As String
    Public Overridable Property Strasse As String
    Public Overridable Property PLZ As String
    Public Overridable Property Ort As String
    Public Overridable Property Land As String
    Public Overridable Property EMail As String
    Public Overridable Property UstIDNr As String
    Public Overridable Property Seminare As ICollection(Of Seminar)
End Class
```

Die Entität Seminar

Darauf baut die Klasse für die Entität **Seminar** auf. Sie enthält einen Verweis auf das Seminarthema, auf dessen Basis es erstellt wurde. Außerdem enthält es drei Eigenschaften zum Aufnehmen von **Titel**, **Inhalt** und **Preis** des Seminars, die beim Anlegen eines Objekt auf Basis der Klasse **Seminar** aus dem jeweiligen **Seminarthema**-Objekt gefüllt werden. Dazu gibt es noch das Feld **Seminartermin** und eine Auflistung namens **Kunden**, welche die Kunden enthalten soll, die dieses Seminar gebucht haben. Diese wird in der Konstruktormethode erstellt:

```
Partial Public Class Seminar
    Public Overridable Property ID As Int32
    Public Overridable Property Titel As String
    Public Overridable Property Inhalt As String
    Public Overridable Property Preis As Decimal
    Public Overridable Property Seminartermin As Date
    Public Overridable Property Kunden As ICollection(Of Kunde)
End Class
```

Die dbContext-Klasse SeminarverwaltungContext

Die beim Hinzufügen des Entity Data Models automatisch erstellte Klasse **SeminarverwaltungContext** passen wir wie folgt an.

Seminarverwaltung II: Ribbon und Frame

Um eine Anwendung wie eine Seminarverwaltung ergonomisch steuern zu können, benötigen wir die Möglichkeit, alle wichtigen Elemente der Benutzeroberfläche und Funktionen schnell aufzurufen. Dazu verwenden wir ein Ribbon, dass alle benötigten Steuerelemente oben im Anwendungsfenster anbietet. Dieser Artikel zeigt, wie Sie das Ribbon für diese Anwendung definieren und wie Sie die einzelnen Funktionen vom Ribbon aus aufrufen.

Um ein Ribbon zum Hauptfenster der Anwendung hinzuzufügen, benötigen wir zuerst das **Ribbon**-Element. Dieses steht allerdings nicht standardmäßig zur Verfügung, sodass wir noch einen Verweis zum Projekt hinzufügen. Dazu betätigen wir als Erstes den Menüeintrag **Projekt|Verweis hinzufügen...**, was den Dialog **Verweis-Manager** öffnet. Hier suchen wir mit dem Suchfeld oben rechts nach dem Eintrag **Ribbon**, was direkt den Verweis **System.Windows.Controls.Ribbon** einblendet. Diesen markieren wir und schließen den Dialog mit einem Klick auf die Schaltfläche **OK** (siehe Bild 1).

Ribbon anlegen

Danach finden wir im XAML-Code für das Hauptfenster per IntelliSense schnell das **Ribbon**-Element, mit dem wir die Definition beginnen. Der Code für das Ribbon und das Frame-Element, in dem wir die einzelnen Seiten anzeigen wollen, sieht wie folgt aus. Als Erstes definieren wir ein paar **Grid**-Zeilen, um das Ribbon und das **Frame**-Element in jeweils einer Zeile unterzubringen:

```
<Window x:Class="MainWindow" ...
        Title="MainWindow" Height="450" Width="800" WindowStartupLocation="CenterScreen">
    <Grid>
        <Grid.ColumnDefinitions>
```

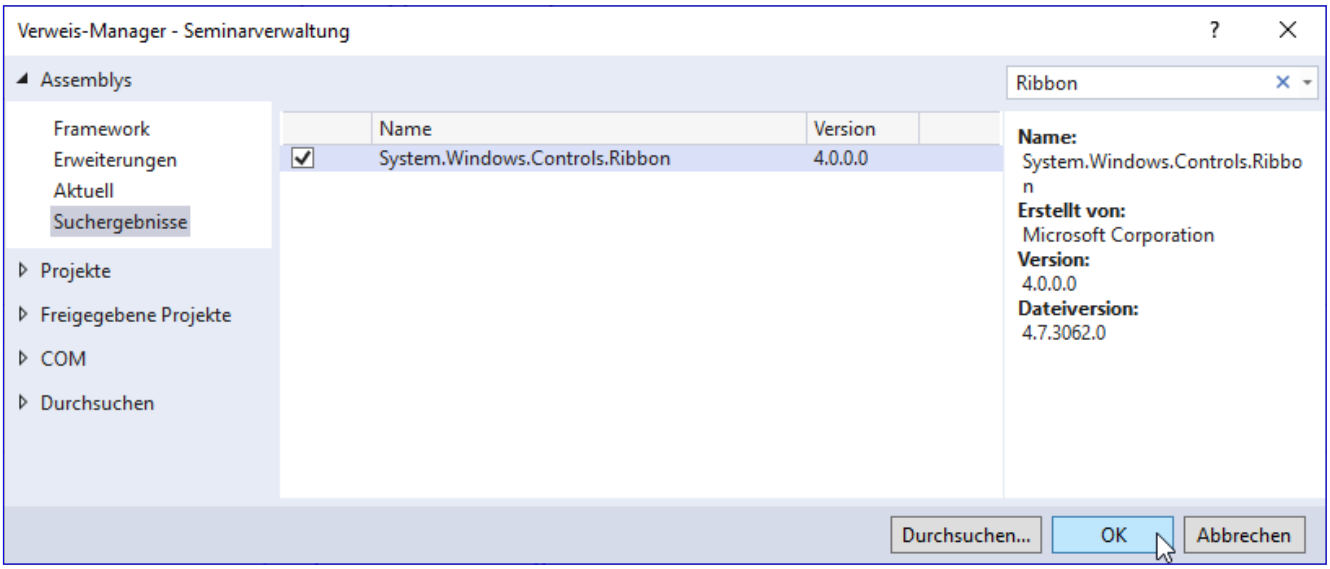


Bild 1: Auswahl des Verweises für die Ribbon-Definition

```
        <ColumnDefinition></ColumnDefinition>
    </Grid.ColumnDefinitions>
    <Grid.RowDefinitions>
        <RowDefinition Height="Auto"></RowDefinition>
        <RowDefinition Height="*"></RowDefinition>
    </Grid.RowDefinitions>
```

Dann folgt die Definition des **Ribbon**-Elements, dem wir ein **RibbonTab**- und einige **RibbonGroup**-Elemente hinzufügen. Die **RibbonGroup**-Elemente enthalten ein oder mehrere **RibbonButton**-Elemente. Für diese haben wir jeweils die Attribute **x:Name**, **Label**, **LargeImageSource** und **Click** definiert:

```
    <Ribbon Grid.Row="0">
        <RibbonTab Header="Seminarverwaltung">
            <RibbonGroup Header="Kunden">
                <RibbonButton x:Name="btnKundenuebersicht" Label="Übersicht"
                              LargeImageSource="images/users_crowd.png"
                              Click="btnKundenuebersicht_Click"></RibbonButton>
                <RibbonButton x:Name="btnKundeAnlegen" Label="Anlegen"
                              LargeImageSource="images/user_add.png"
                              Click="btnKundeAnlegen_Click"></RibbonButton>
            </RibbonGroup>
            <RibbonGroup Header="Seminare">
                <RibbonButton x:Name="btnSeminaruebersicht" Label="Übersicht"
                              LargeImageSource="images/lecture.png"
                              Click="btnSeminaruebersicht_Click"></RibbonButton>
                <RibbonButton x:Name="btnSeminarAnlegen" Label="Anlegen"
                              LargeImageSource="images/book_plus.png"
                              Click="btnSeminarAnlegen_Click"></RibbonButton>
            </RibbonGroup>
            <RibbonGroup Header="Referenten">
                <RibbonButton x:Name="btnReferentenuebersicht" Label="Übersicht"
                              LargeImageSource="images/teacher_blackboard.png"
                              Click="btnReferentenuebersicht_Click"></RibbonButton>
                <RibbonButton x:Name="btnReferentAnlegen" Label="Anlegen"
                              LargeImageSource="images/teacher_blackboard_plus.png"
                              Click="btnReferentAnlegen_Click"></RibbonButton>
            </RibbonGroup>
            <RibbonGroup Header="Bestellungen">
                <RibbonButton x:Name="btnImportieren" Label="Importieren"
                              LargeImageSource="images/shopping_basket_full.png"
                              Click="btnImportieren_Click"></RibbonButton>
```

Seminarverwaltung III: Daten vom Shopsystem

Unsere Seminarverwaltung lebt unter anderem davon, dass Bestellungen über ein Shopsystem eingehen. Hier nutzen wir elopage, einen Anbieter, mit dem man verschiedene digitale Dienstleistungen online verkaufen kann. elopage übernimmt dabei einige Aufgaben wie etwa die Bereitstellung von Produkt- und Bestellseiten. Nachdem eine Bestellung erfolgt ist, landen die Daten des Kunden inklusive der Daten zum bestellten Artikel in der Datenbank von elopage. Die Bestelldaten können wir per CSV-Datei exportieren und in unsere Seminarverwaltung einlesen. Dazu verwenden wir das NuGet-Paket CsvHelper, das wir in zwei weiteren Artikeln bereits vorgestellt haben. Der vorliegende Artikel zeigt, wie wir die beschriebenen Techniken für den Import in einem konkreten Anwendungsfall nutzen können.

Schritt für Schritt

Wir wollen den Import der CSV-Datei in die einzelnen Tabellen der Anwendung in mehreren Schritten vollziehen:

- Als Erstes wollen wir die Daten aus der CSV-Datei in einer einzigen Klasse verfügbar machen, damit wir komfortabler auf die enthaltenen Daten zugreifen können. Dazu erstellen wir eine Klasse, die für jede Spalte der CSV-Datei ein Feld enthält. Das wird hier und da etwas aufwendiger, wie wir gleich sehen werden.

- Die einzelnen Objekte mit den Daten aus jeweils einer Zeile fügen wir zu einer Auflistung hinzu.

- Danach durchlaufen wir die Auflistung mit allen Datensätzen und lesen daraus die Informationen aus, die wir den einzelnen Entitäten unseres Entity Data Models zuweisen wollen. Diese legen wir dann an und speichern die enthaltenen Daten in der zugrunde liegenden Datenbank.

Herausfordernde CSV-Datei

Die von elopage exportierte Datei liefert uns einige Herausforderungen. Sie sieht wie in Bild 1 aus.

Die Herausforderung ist, dass wir die Felder möglichst so wie in der Kopfzeile dargestellt in Objekten einer neuen Klasse namens **ImportBestellung** abbilden wollen – um möglichst ein 1:1-Mapping zu realisieren:

```
TOKEN;PRODUKT;ERSTELLT;ZAHLUNGSSTATUS;METHODE;PLAN;TESTZEITRAUM;ANZAHL ZAHLUNGEN;BEZAHLT;PRODUCT ID;FUER
PS;FAELLIGER BETRAG;WAEHRUNG;UNTERNEHMEN;PLAN;ZAHLUNGSPLAN;ZAHLUNGSPLAN ID;GUTSCHEINCODE;CAMPAIGN-ID;MIT
TESTZEITRAUM;FIXER FAELLIGKEITSTAG;PRODUKTNAME;EVENT;ORT;EVENT-DATUM;TICKETANZAHL;TICKET CODE;VORNA-
ME;NACHNAME;E-MAIL;TELEFON;LAND;STADT;STRASSE;HAUSNUMMER;PLZ;UNTERNEHMEN;UST-IDNR.;EMPFAENGER NAME;EMP-
FAENGER E-MAIL-ADRESSE;EMPFAENGER TELEFON;EMPFAENGER LAND;EMPFAENGER STADT;EMPFAENGER STRASSE;EMPFAENGER
HAUSNUMMER;EMPFAENGER PLZ;EMPFAENGER FIRMA;NUTZER;PUBLISHER ID;AELTESTE BEZAHLTE RATE;FAELLIGKEITSDA-
TUM;BESTELL-ID;"";"";"";"";"";"";"";"";"";"";"";"";"";"";"";"";"";""
```

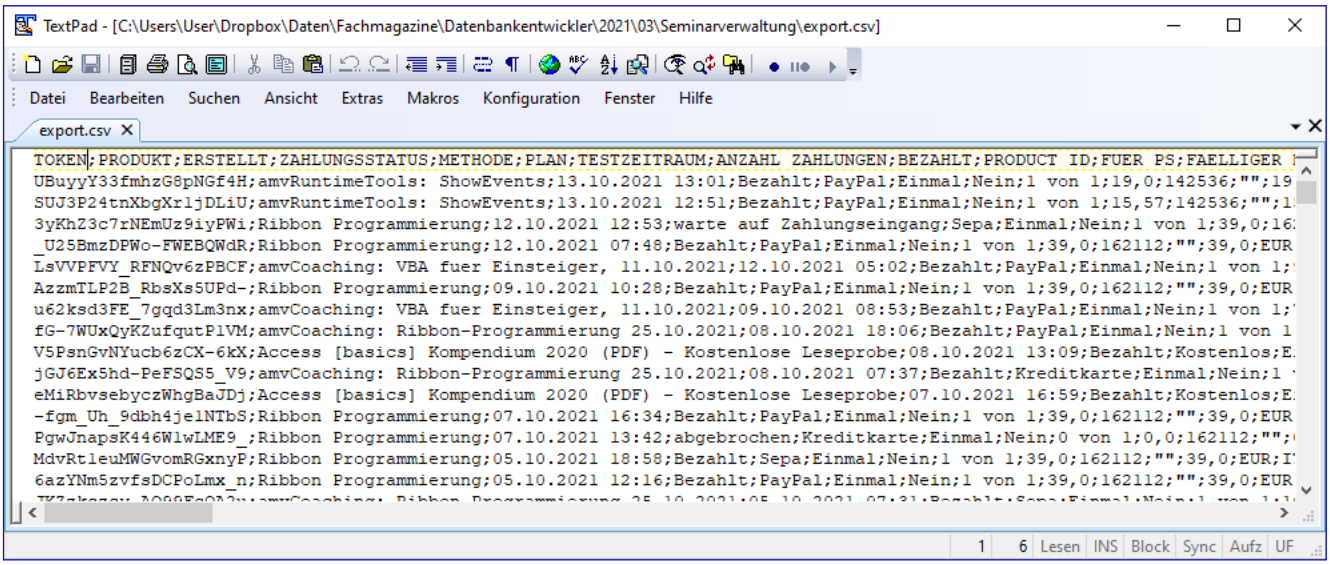


Bild 1: Der CSV-Export des Shopsystems

Sie sehen hier aber direkt einige Elemente mit Namen, die so nicht als Eigenschaften einer .NET-Klasse verwendet werden können. Einige der Spaltenüberschriften enthalten Leerzeichen (zum Beispiel **PRODUCT ID** oder **FAELLIGER BETRAG**), andere entsprechen reservierten Begriffen (**EVENT**), wieder andere kommen doppelt vor (**UNTERNEHMEN**). Zum Glück bietet das **CsvHelper**-Paket Möglichkeiten, solche Felder über entsprechende Data Annotations zuzuordnen.

Wir haben im Folgenden die Klasse **ImportBestellung** ausschnittsweise abgebildet und vor allem die Spezialfälle aufgeführt. Die Klasse verwendet einige zusätzliche Namespaces:

```
Imports CsvHelper
Imports CsvHelper.Configuration
Imports CsvHelper.Configuration.Attributes
```

Die Klasse selbst sieht wie folgt aus:

```
Public Class ImportBestellung
    Public Property TOKEN As String
    ...
    <Name("PLAN")>
    <NameIndex(0)>
    Public Property PLAN1 As String 'Kommt zwei Mal vor
    Public Property TESTZEITRAUM As String
    <Name("ANZAHL ZAHLUNGEN")>
    Public Property ANZAHLZAHLUNGEN As String 'Enthält Leerzeichen
    <Name("PRODUCT ID")>
```

```
    Public Property PRODUCT_ID As String 'Enthält Leerzeichen
    <Name("FUER PS")>
    ...
    <Name("UNTERNEHMEN")>
    <NameIndex(0)>
    Public Property UNTERNEHMEN As String 'Kommt zwei Mal vor.
    <Name("PLAN")>
    <NameIndex(1)>
    Public Property PLAN2 As String
    Public Property ZAHLUNGSPLAN As String
    <Name("EVENT")>
    Public Property Event1 As String 'Event ist ein reserviertes Schlüsselwort
    <Name("EVENT-DATUM")>
    Public Property Event_DATUM As String 'Enthält Minuszeichen
    <Name("UNTERNEHMEN")>
    <NameIndex(1)>
    Public Property UNTERNEHMEN_1 As String 'Zweites Vorkommen
    <Name("UST-IDNR.")>
    ...
End Class
```

Die vollständige Klasse finden Sie im Beispielprojekt zu diesem Artikel.

Importieren der CSV-Datei

Das Importieren der CSV-Datei wollen wir möglichst komfortabel gestalten, deshalb speichern wir das zuletzt verwendete Verzeichnis in einer Option der Anwendung und rufen dieses beim nächsten Anzeigen des Dateiauswahl-Dialogs wieder ab.

Für das Importieren haben wir dem Ribbon eine Schaltfläche namens **btnImportieren** hinzugefügt. Diese löst die folgende Ereignismethode aus:

```
Private Sub btnImportieren_Click(sender As Object, e As RoutedEventArgs)
    Dim strDateiname As String
    Dim objImportBestellungen As List(Of ImportBestellung)
    strDateiname = DateinameErmitteln()
    objImportBestellungen = CSVImport(strDateiname)
    CSVImportVerarbeiten(objImportBestellungen)
End Sub
```

Hier rufen wir zuerst die Funktion **DateinameErmitteln** auf. Diese liest mit der Funktion **VerzeichnisLesen** das zuletzt verwendete Verzeichnis in die Variable **strImportverzeichnis** ein. Ist diese leer, verwendet die Methode

das Verzeichnis der **.exe**-Datei der Anwendung als Startverzeichnis für die Eigenschaft **InitialDirectory**. Anderenfalls kommt das Verzeichnis aus **strImportverzeichnis** zum Einsatz:

```
Private Function DateinameErmitteln() As String
    Dim strImportpfad As String
    Dim strImportverzeichnis As String
    Dim objOpenFileDialog As New OpenFileDialog
    strImportverzeichnis = VerzeichnisLesen()
    If strImportverzeichnis.Length = 0 Then
        objOpenFileDialog.InitialDirectory = AppDomain.CurrentDomain.BaseDirectory
    Else
        objOpenFileDialog.InitialDirectory = strImportverzeichnis
    End If
```

Dann stellt die Methode **.csv** als Dateiendung für den **Datei öffnen**-Dialog ein. Die Methode **ShowDialog** zeigt den Dialog an. Wird dieser unter Angabe eines Dateinamens geschlossen, weist die Methode diesen der Variablen **strImportpfad** zu. Schließlich sorgt die Methode **VerzeichnisAktualisieren** noch dafür, dass das gewählte Verzeichnis in die Optionen der Anwendung geschrieben wird:

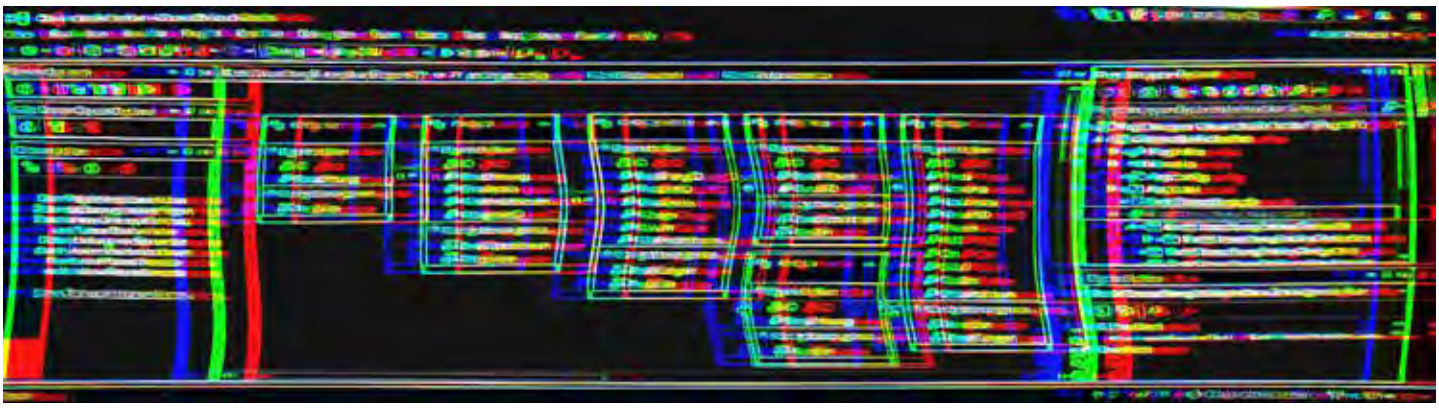
```
    objOpenFileDialog.DefaultExt = ".csv"
    If (objOpenFileDialog.ShowDialog = True) Then
        strImportpfad = objOpenFileDialog.FileName
        VerzeichnisAktualisieren(strImportpfad)
    End If
    Return strImportpfad
End Function
```

Speichern des Importverzeichnisses

AUSGABE 06/2021

DATENBANK ENTWICKLER

MAGAZIN FÜR DIE DATENBANKENTWICKLUNG MIT VISUAL STUDIO FÜR DESKTOP, WEB UND CO.



TOP-THEMEN:

WPF CONTROLS	TabControl: Das Registersteuerelement von WPF	SEITE 3
WPF CONTROLS	Registerseiten im TabControl an Daten binden	SEITE 13
VB-GRUNDLAGEN	Unit-Testing mit Visual Studio	SEITE 21
LÖSUNGEN	Seminarverwaltung Teil IV	SEITE 47

André Minhorst Verlag

TabControl: Das Registersteuerelement von WPF

Genau wie unter Microsoft Access finden Sie auch unter WPF ein Registersteuerelement – das sogenannte TabControl-Element. Sie können dieses Steuerelement vielseitig einsetzen, um beispielsweise Inhalte, die sonst nicht auf einer Bildschirmseite Platz finden, dennoch halbwegs übersichtlich darzustellen. Dieser Artikel stellt das Registersteuerelement von WPF vor und zeigt, wie Sie es in Ihren eigenen Anwendungen nutzen können.

Das leere TabControl-Element

Wenn wir einem Grid in einem XAML-Fenster ein **TabControl**-Element hinzufügen, bekommen wir auf jeden Fall wesentlich weniger geboten, als wenn wir ein solches beispielsweise einem Access-Formular hinzufügen:

```
<Grid>
    <TabControl Margin="5"></TabControl>
</Grid>
```

Wir müssen diesem schon per **Margin**-Attribut einen Abstand zum Fensterrahmen hinzufügen, damit wir erkennen können, dass es überhaupt vorhanden ist (siehe Bild 1).

Bild 1: Ein leeres **TabControl**-Element

Ein TabItem-Element hinzufügen

Erst wenn wir dem **TabControl**-Element ein **TabItem**-Element unterordnen, erkennen wir, dass es sich bei dem angezeigten Rahmen um ein **TabControl**-Element handelt:

```
<TabControl Margin="5">
    <TabItem Header="Seite 1"></TabItem>
</TabControl>
```

Dem **TabItem**-Element weisen wir über das **Header**-Attribut den Text für den Registerreiter zu (siehe Bild 2).

Bild 2: Ein **TabControl**-Element mit einem **TabItem**-Element

Auf die gleiche Weise legen Sie weitere Registerreiter an, die wie in Bild 3 erscheinen:

```
<TabControl Margin="5">
    <TabItem Header="Seite 1"></TabItem>
```

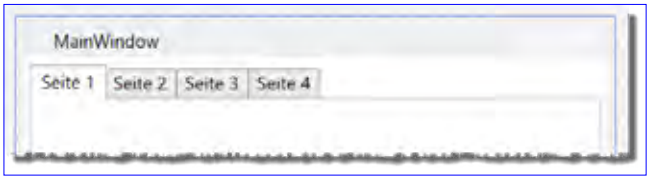


Bild 3: Mehrere **TabItem**-Elemente

```
    <TabItem Header="Seite 2"></TabItem>
    <TabItem Header="Seite 3"></TabItem>
    <TabItem Header="Seite 4"></TabItem>
</TabControl>
```

Um im Entwurf der XAML-Seite zu einer anderen Registerseite als der ersten zu wechseln, klicken Sie entweder auf den jeweiligen Registerreiter – also so, wie Sie es auch später in der laufenden Anwendung erledigen würden – oder Sie positionieren die Einfügemarke auf einem der **TabItem**-Elemente im XAML-Code.

Steuerelemente zu einem TabItem-Element hinzufügen

Allerdings können wir so nicht erkennen, ob tatsächlich der Inhalt der jeweiligen Registerseite angezeigt wird. Um dies zu erreichen, fügen wir jedem **TabItem**-Element zunächst ein **Label**-Steuerelement mit einer passenden Beschriftung hinzu:

```
<TabItem Header="Seite 1">
    <Label>Dies ist die erste Seite.</Label>
</TabItem>
<TabItem Header="Seite 2">
    <Label>Dies ist die zweite Seite.</Label>
</TabItem>
<TabItem Header="Seite 3">
    <Label>Dies ist die dritte Seite.</Label>
</TabItem>
<TabItem Header="Seite 4">
    <Label>Dies ist die vierte Seite.</Label>
</TabItem>
```

Bild 4: **TabItem**-Element mit Steuerelement

Ein Klick auf die zweite Registerseite zeigt wie erwartet das zu dieser Seite hinzugefügte **Label**-Steuerelement an (siehe Bild 4).

Nur ein Element je TabItem-Element

Dass es möglich ist, einfach Steuerelemente im **TabItem**-Element zu platzieren, liegt daran, dass das **TabItem**-Steuerelement vom **ContentControl**-Element abstammt. Das bedeutet gleichzeitig, dass Sie in jedem **TabItem**-Element aber auch nur ein einziges Steuerelement unterbringen dürfen. Wenn Sie wie in Bild 5 noch ein **Text-**

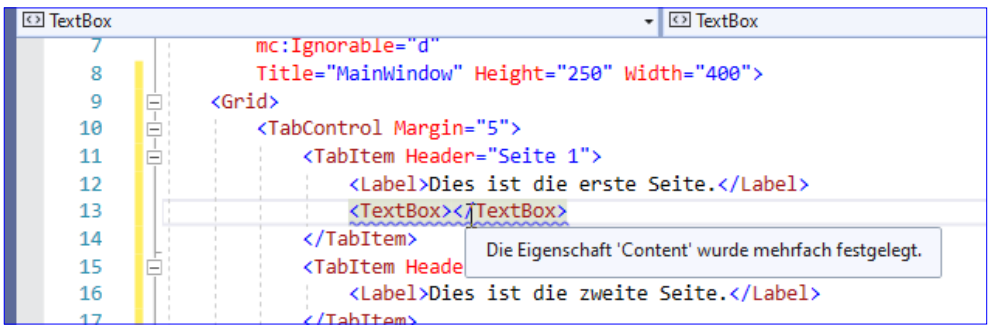


Bild 5: Das **TabItem**-Element kann nur ein Unterelement aufnehmen.

Box-Steuerelement hinzufügen möchten, erhalten Sie dementsprechend direkt eine Fehlermeldung. Das ist allerdings kein Problem, denn auch einem **Window**-Element, also dem Hauptelement eines XAML-Fensters, können Sie ja nur ein Element unterordnen. Aber wozu gibt es Elemente, die zum Organisieren mehrerer untergeordneter Elemente vorgesehen sind wie **Grid**, **StackPanel** und so weiter?

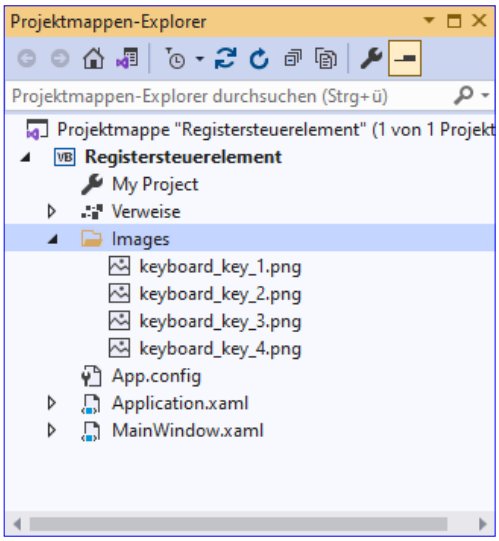


Bild 6: Images für die Registerreiter

Registerreiter anpassen

Wenn Sie von Access kommen, wird Ihnen allein die Idee, die Registerreiter anzupassen, ungewohnt vorkommen – dort war das höchste der Gefühle, den angezeigten Text und die Schriftgröße und -art einzustellen. Unter WPF bieten sich naturgemäß einige zusätzliche Möglichkeiten. Wie wäre es beispielsweise mit ein paar Icons, mit denen der Benutzer noch schneller die gewünschte Registerseite finden kann? Kein Problem: Fügen wir den Registerreitern Icons hinzu!

Registerreiter mit Icons

Um die Registerreiter mit Icons auszustatten, müssen wir dem Projekt zunächst entsprechende Ressourcen hinzufügen. Für diese legen wir einen Ordner namens **Images** an. In diesen ziehen Sie die gewünschten Bilddateien, sodass der Projektmappen-Explorer anschließend wie in Bild 6 aussieht.

Danach passen Sie den Code des **TabItem**-Elements an, indem Sie das **Header**-Attribut, das ja nun nicht mehr nur einen Text aufnehmen soll, sondern ein Icon und einen Text, entfernen und stattdessen ein Property-Element namens **TabItem.Header** zum **TabItem**-Element hinzufügen. Wie schon beim **TabItem**-Element gesehen, können Sie auch dem Property-Element **TabItem.Header** nur ein einziges Element unterordnen. Allerdings kann es sich hierbei auch beispielsweise um ein **StackPanel**-Element handeln, mit dem wir das Icon und die Beschriftung nebeneinander anordnen können. Dafür ist die Einstellung des Attributs **Orientation** des **StackPanel**-Elements auf den Wert **Horizontal** notwendig.

Die Definition für den ersten Registerreiter sieht dann wie folgt aus:

```
<TabControl Margin="5">
    <TabItem>
        <TabItem.Header>
            <StackPanel Orientation="Horizontal">
                <Image Source="Images\keyboard_key_1.png"></Image>
                <Label>Seite 1</Label>
            </StackPanel>
        </TabItem.Header>
        <Label>Dies ist die erste Seite.</Label>
```

```
    </TabItem>
    ...
</TabControl>
```

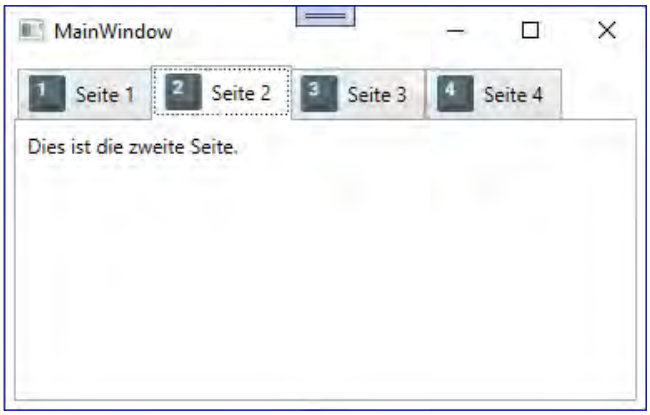


Bild 7: Registerreiter mit Icons

Wenn wir alle vier Registerreiter mit den passenden Icons ausstatten, sieht dies schließlich wie in Bild 7 aus.

Hier gibt es noch viel mehr Möglichkeiten – eigentlich können Sie alle möglichen Elemente zu einem Registerreiter beziehungsweise zu einem **TabItem**-Element hinzufügen. Ob das sinnvoll ist, sei dahingestellt, aber die Anzeige mit Icons und Beschriftung sieht bereits recht professionell aus.

Das TabControl mit Visual Basic programmieren

In den folgenden Abschnitten schauen wir uns verschiedene Möglichkeiten für den Zugriff auf das **TabControl**-Element und seine Steuerung an. Dabei behandeln wir die folgenden Themen:

- Ermitteln des Index der aktuellen Seite des **TabControl**-Elements
- Eine bestimmte Seite im **TabControl**-Element einstellen
- Weitere Informationen über das selektierte **TabItem**-Element ermitteln
- Ereignis beim Wechsel des **TabItem**-Elements
- Kein **TabItem**-Element markieren
- **TabItem**-Elemente per Code hinzufügen
- **TabItem**-Elemente per Code entfernen

Für diese Beispiele haben wir das Fenster ein wenig umstrukturiert. Wir haben dem Grid zwei **RowDefinition**-Elemente hinzugefügt, von denen das obere nun das **TabControl**-Element aufnimmt und das untere ein **StackPanel**-Element, dem wir im weiteren Verlauf einige Schaltflächen hinzufügen. Für das **TabControl**-Element haben wir außerdem den Namen **tab** eingestellt:

```
<TabControl x:Name="tab" Grid.Row="0">
```

Das **StackPanel** in der unteren Zeile des **Grid**-Elements enthält zunächst eine Schaltfläche, zu der aber noch weitere hinzukommen werden:

Registerseiten im TabControl an Daten binden

Im Artikel »TabControl: Das Registersteuerelement von WPF« haben wir uns die grundlegenden Eigenschaften des Registersteuerelements von .NET und WPF angesehen. Dieses bietet ähnliche Möglichkeiten wie das Registersteuerelement, das Sie vielleicht von Microsoft Access kennen. An einigen Stellen gehen die Möglichkeiten, bedingt durch die größere Flexibilität der Beschreibungssprache XAML, darüber hinaus. So können Sie beispielsweise das TabControl und seine TabItem-Elemente an eine Datenquelle wie eine ObservableCollection binden und die Daten der Objekte dieser Collection sowohl für die Gestaltung der Registerreiter als auch für den eigentlich Inhalt einer jeden Registerseite nutzen.

Unter Microsoft Access müssen Sie, wenn Sie ein Registersteuerelement mit je einer Registerseite pro Datensatz versehen wollen, jede Registerseite per VBA-Code hinzufügen. Das gelingt zwar, aber unter .NET/WPF ist das wesentlich komfortabler zu erledigen.

Dazu legen Sie ein neues WPF-Projekt mit Visual Basic als Sprache an. Die nachfolgenden Elemente fügen wir dem Hauptfenster **MainWindow.xaml** beziehungsweise der dazu gehörenden Code behind-Klasse **MainWindow.xaml.vb** hinzu.

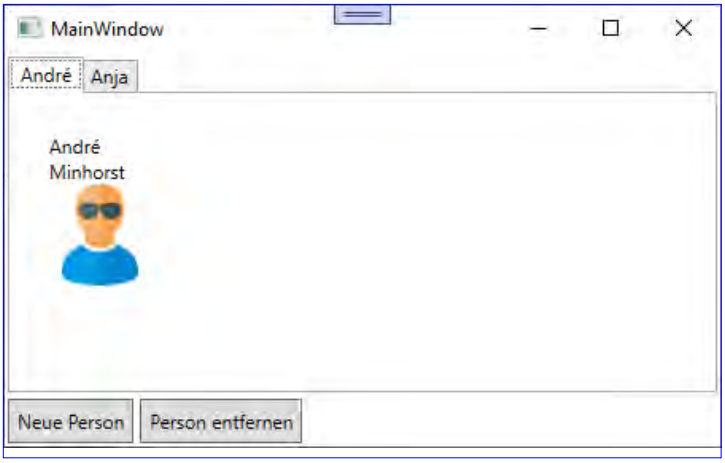


Bild 1: So soll das **TabControl**-Element aussehen

Das Ergebnis dieses Artikels soll wie in Bild 1 aussehen.

Personen-Auflistung als Datenquelle für ein TabControl

Unser Vorhaben ist es, die Elemente einer Klasse namens Person als Datenquelle für ein **TabControl**-Element zu nutzen. Diese Klasse haben wir wie folgt beschrieben:

```
Public Class Person
    Public Property ID As Long
    Public Property Vorname As String
    Public Property Nachname As String
    Public Property Bild As String
End Class
```

Der Datei **MainWindow.xaml.vb** fügen wir außerdem per **Imports**-Anweisung den Namespace **System.Collections.ObjectModel** hinzu, der die für dieses Beispiel benötigte **ObservableCollection**-Klasse beisteuert:

```
Imports System.Collections.ObjectModel
```

In der **MainWindow**-Klasse schließlich deklarieren wir zunächst die Property für die **Personen**-Liste. Diese deklarieren wir als Public, da sie als Datenquelle für das TabControl-Element dienen soll. Wir legen mit **Person** auch gleich den Typ der aufzunehmenden Objekte fest:

```
Public Class MainWindow
    Public Property Personen As ObservableCollection(Of Person)
```

Auflistung der anzuzeigenden Personen füllen

Im Konstruktor der Klasse erledigen wir dann Aufgaben, die wir schon in vielen Artikeln durchgeführt haben. Wir deklarieren ein Objekt der Klasse **Person** und initialisieren das **ObservableCollection**-Element **Personen**:

```
    Public Sub New()
        Dim Person As Person
        Personen = New ObservableCollection(Of Person)
```

Danach erstellen wir zwei **Person**-Elemente und weisen diesen die entsprechenden Eigenschaften zu. Für die Eigenschaft **Bild** geben wir den Pfad von Bilddateien an, die wir im Projektmappen-Explorer in einem separaten Ordner namens **Images** gespeichert haben.

Damit wir diese so wie hier mit **\Images\user_sunglasses.png** referenzieren können, müssen Sie für die Bilddateien die Eigenschaft **In Ausgabeverzeichnis kopieren** auf **Immer kopieren** oder **Kopieren, wenn neuer** einstellen (siehe Bild 2).

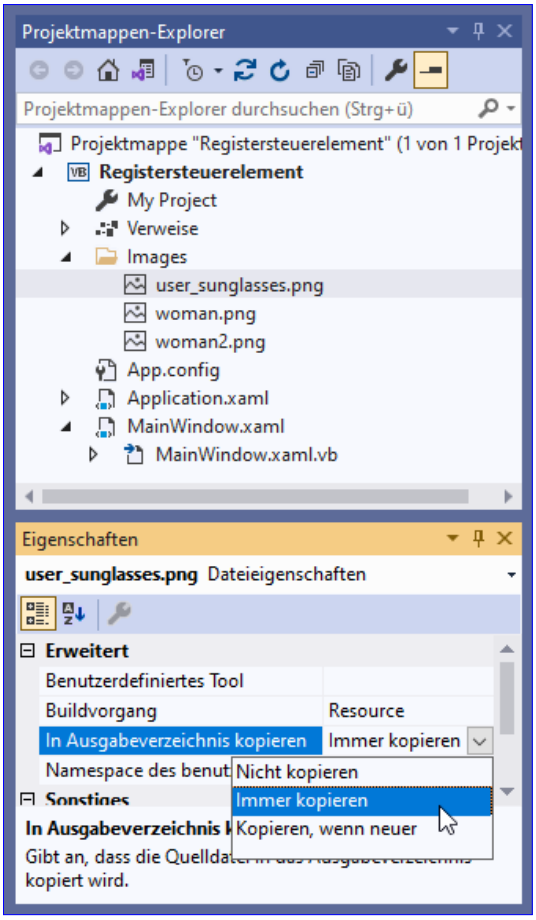


Bild 2: Einstellung, damit die Bilddateien ins Ausgabeverzeichnis kopiert werden

Der Code zum Anlegen der beiden vorerst benötigten **Person**-Elemente und zum Hinzufügen der Elemente mit der **Add**-Methode zur **Personen**-Liste sieht so aus:

```
        Person = New Person
        With Person
            .ID = 1
            .Vorname = "André"
            .Nachname = "Minhorst"
            .Bild = "\Images\user_sunglasses.png"
        End With
```

```
        Personen.Add(Person)
        Person = New Person
        With Person
            .ID = 2
            .Vorname = "Anja"
            .Nachname = "Minhorst"
            .Bild = "\Images\woman2.png"
        End With
        Personen.Add(Person)
```

Schließlich rufen wir noch die Methode **InitializeComponent** auf, damit die XAML-Definition angewendet wird, und stellen die Eigenschaft **DataContext** des Fensters auf das Code behind-Modul ein:

```
        InitializeComponent()
        DataContext = Me
    End Sub
```

Dies zeigt schön, wie die Trennung der Definition der Benutzeroberfläche und der Anwendungslogik unter WPF funktioniert. Wir haben noch keine Elemente zur Benutzeroberfläche hinzugefügt und die Anwendungslogik ist quasi schon fertig. Nun können wir uns an den Entwurf der Benutzeroberfläche begeben.

Layout der Elemente vorbereiten

Wie oben im Bild zu erkennen, wollen wir nicht nur das **TabControl**-Element, sondern auch noch zwei Schaltflächen hinzufügen, mit denen wir neue Elemente zum **TabControl**-Element hinzufügen oder vorhandene Elemente entfernen können. Damit die Schaltflächen und das **TabControl**-Element nicht genau am Rand und aneinander kleben, definieren wir **Margin**- und **Padding**-Attribute für alle Elemente dieser Typen:

```
<Window x:Class="MainWindow" ... Title="MainWindow" Height="250" Width="400">
    <Window.Resources>
        <Style TargetType="Button">
            <Setter Property="Margin" Value="2"></Setter>
            <Setter Property="Padding" Value="5"></Setter>
        </Style>
        <Style TargetType="TabControl">
            <Setter Property="Margin" Value="2"></Setter>
            <Setter Property="Padding" Value="5"></Setter>
        </Style>
    </Window.Resources>
```

Dann definieren wir ein **Grid**-Element mit zwei Zeilen, von denen die obere das **TabControl**-Element und das untere ein **StackPanel**-Element mit den Schaltflächen aufnehmen soll:

```
<Grid>
    <Grid.RowDefinitions>
        <RowDefinition Height="*"></RowDefinition>
        <RowDefinition Height="Auto"></RowDefinition>
    </Grid.RowDefinitions>
```

TabControl-Element hinzufügen

Schließlich folgt das **TabControl**-Element. Dieses sieht in der nachfolgenden Definition nur in der ersten Zeile ungefähr so aus wie das mit statischen Daten gefüllte Registersteuerelement aus dem Artikel **TabControl: Das Registersteuerelement von WPF** (**www.datenbankentwickler.net/290**). Aber schon die erste Zeile verwendet das **ItemsSource**-Attribut und weist diesem eine Bindung an die Eigenschaft **Personen** zu, welche das **ObservableCollection**-Element enthält.

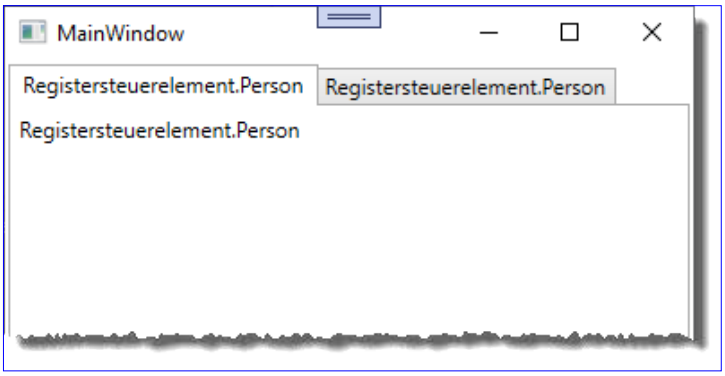


Bild 3: **TabControl**-Element nur mit Bindung, aber noch ohne Inhalte

Sie können das Projekt bereits jetzt einmal starten und finden dann die Ansicht aus Bild 3 vor. Die Bindung funktioniert also – die Registerseiten sind bereits jetzt auf **Person**-Elemente eingestellt:

```
<TabControl x:Name="tab" ItemsSource="{Binding Personen}">
    ...
</TabControl>
```

Header des TabItem-Elements füllen

Allerdings wollen wir im Header der Registerseiten den Vornamen der Person anzeigen und auf der jeweiligen Registerseite die Inhalte aller Felder der **Person**-Elemente. Für die Anzeige des Vornamens im Header fügen wir ein Property-Element namens **TabControl.ItemsContainerStyle** hinzu. Dieses erhält ein **Style**-Element, das sich auf die **TabItem**-Elemente beziehen soll. Der Name der Eigenschaft, die wir in diesem Style definieren wollen, heißt **HeaderTemplate**. Der Wert ist ein **DataTemplate**-Element, in dem wir angeben, dass der Header ein **StackPanel**-Element mit einem **TextBlock**-Element enthalten soll. Dieser ist über das Attribut **Text** an die Eigenschaft **Vorname** der übergeordneten Datenquelle gebunden:

```
<TabControl.ItemContainerStyle>
    <Style TargetType="TabItem">
        <Setter Property="HeaderTemplate">
            <Setter.Value>
                <DataTemplate>
                    <StackPanel>
```

Unit-Testing mit Visual Studio

In den bisherigen Ausgaben von Datenbankentwickler haben wir ohne moderne Entwurfsmuster programmiert und beispielsweise den Code von Fenstern im Code behind-Modul gespeichert. Das wollen wir nun ändern und das Entwurfsmuster Model-View-ViewModel (kurz MVVM) vorstellen. Das ermöglicht durch eine Zwischenschicht zwischen der Benutzeroberfläche und dem Entity Data Model das automatisierte Testen der meisten Funktionen einer Anwendung. Durch automatisiertes Testen, hier Unit-Testing genannt, können wir Tests für unseren Code definieren, die wir immer wieder per Mausklick durchführen können. So können Sie Änderungen am Code durchführen und sich blitzschnell versichern, dass der Code noch so läuft wie gewünscht. Im vorliegenden Artikel erläutern wir, was es mit Unit-Testing auf sich hat und wie Sie es unter Visual Studio einsetzen.

Tests von Softwareanwendungen sind meist aufwendig, vor allem, wenn sie manuell durchgeführt werden. Das größte Problem dabei ist: Wenn sich eine Funktion der Anwendung ändert, muss man normalerweise zumindest alle damit zusammenhängenden Funktionen erneut testen, was Zeit und Geld kostet. Eine Automatisierung der Tests über die Benutzeroberfläche ist ebenfalls kritisch zu sehen, was den Aufwand betrifft, denn auch hier können kleinste Änderungen dazu führen, dass die Tests nicht mehr funktionieren und ebenfalls angepasst werden müssen.

Wenn Sie mit einem Entwurfsmuster wie dem eingangs erwähnten MVVM arbeiten, wobei es eine Schicht gibt, welche die Anwendungslogik enthält und die Daten der Benutzeroberfläche liefert, aber keine Abhängigkeit von der Benutzeroberfläche hat, können Sie diese Schicht mit der Anwendungslogik auch ohne den Umweg über die Benutzeroberfläche testen. Dazu initialisieren Sie die entsprechenden Objekte und testen deren Methoden.

Weil sich diese Tests meist auf einzelne Klassen oder Komponenten beziehen, werden sie im Allgemeinen Unit-Tests genannt.

Beispielprojekt für die Unit-Tests erstellen

Da wir in diesem Magazin meist Desktop-Anwendungen programmieren, wollen wir auch als Beispiel für die Unit-Tests ein solches Projekt nutzen. Also legen Sie ein neues Projekt des Typs **WPF-App (.NET-Framework)** an und legen für dieses den Namen **UnitTestBeispiel** fest.

Im Anschluss an diesen Vorgang sollte Visual Studio geöffnet sein und im Projektmappen-Explorer das neu erstellte Projekt anzeigen.

Mögliche Konfigurationen für Unit-Tests

Wenn Sie Unittests durchführen wollen, haben Sie verschiedene Möglichkeiten – zum Beispiel die folgenden:

- Sie erstellen ein eigenes Projekt, das nur die Unittests enthält und das Sie gemeinsam mit dem Testprojekt in einer Projektmappe verwalten.

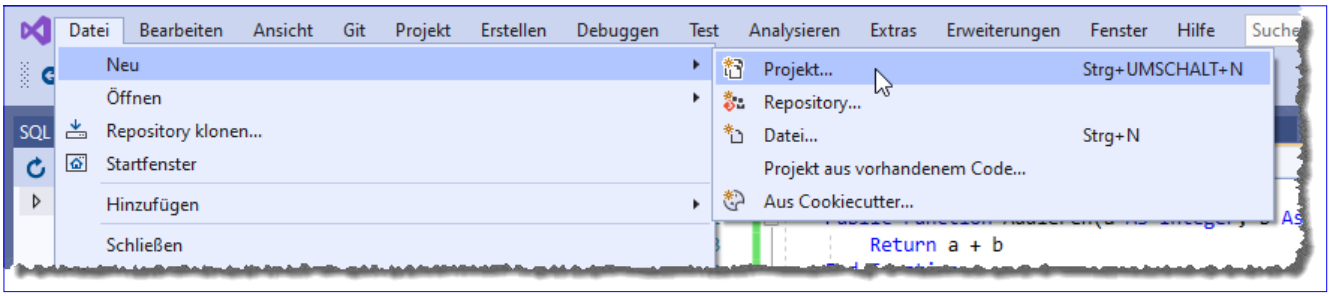


Bild 1: Anlegen eines neuen Projekts

- Sie erstellen die zum Testen notwendigen Elemente innerhalb des zu testenden Projekts.

Eines der Ziele beim Unit-Testing ist, den Code des zu testenden Projekts nicht aufzublähen. Deshalb fügen wir die Unit-Tests auch nicht zu dem Projekt hinzu, das wir testen wollen, sondern legen ein eigenes Projekt dafür an. Dieses Projekt soll allerdings in einer Projektmappe mit dem zu testenden Projekt landen. Deshalb gehen wir wie folgt vor, um das Testprojekt zu erstellen:

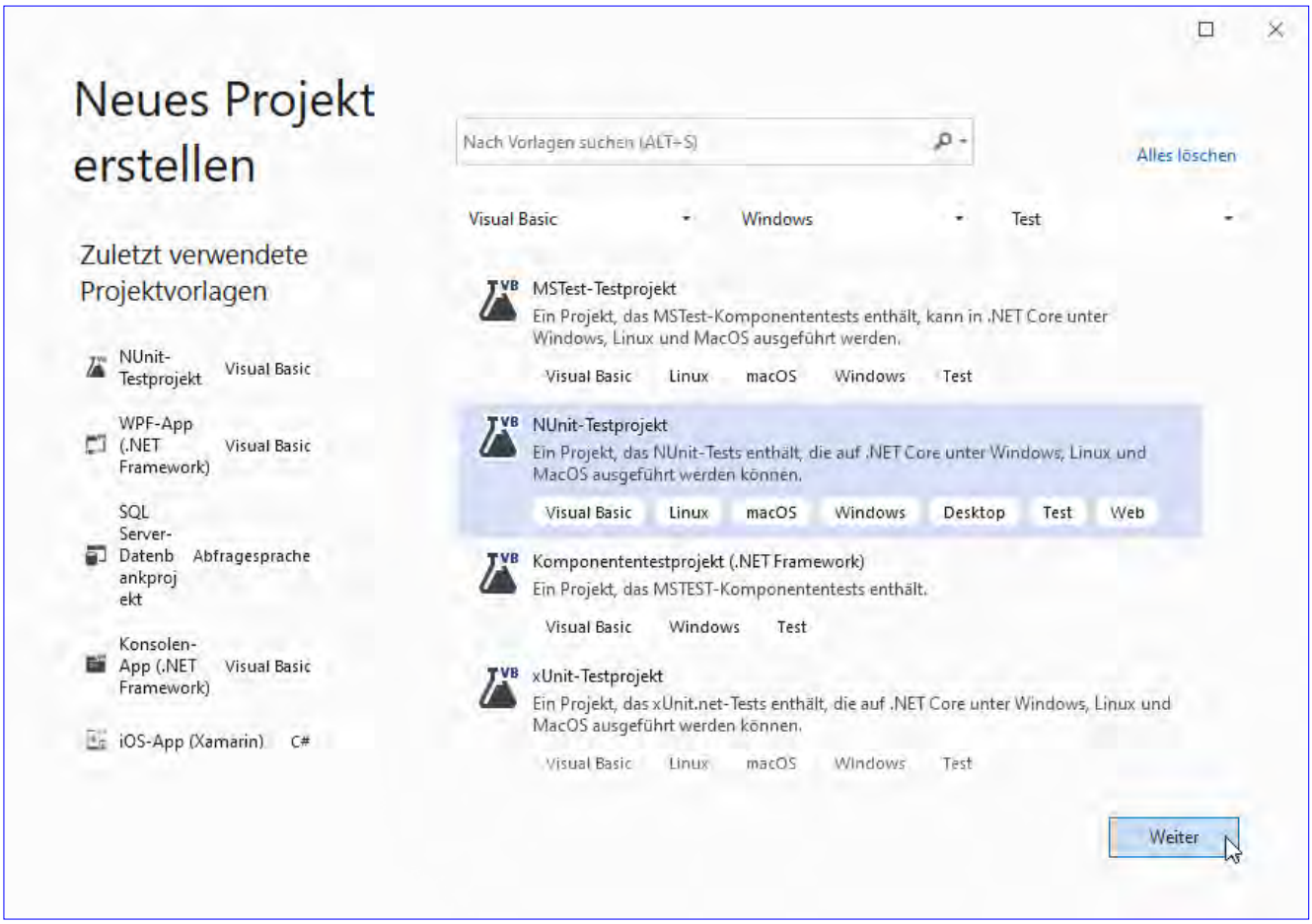


Bild 2: Auswahl des Projekttyps

- Betätigen Sie den Menübefehl **Datei|Neu|Projekt...** (siehe Bild 1).

- Wählen Sie im Dialog **Neues Projekt erstellen** mit dem linken Auswahlfeld den Eintrag **Visual Basic** und mit dem rechten Auswahlfeld den Eintrag **Test** aus und finden Sie die verfügbaren Testprojekte in der Liste. Hier wählen wir die Projektvorlage **NUnit-Testprojekt** aus (siehe Bild 2).

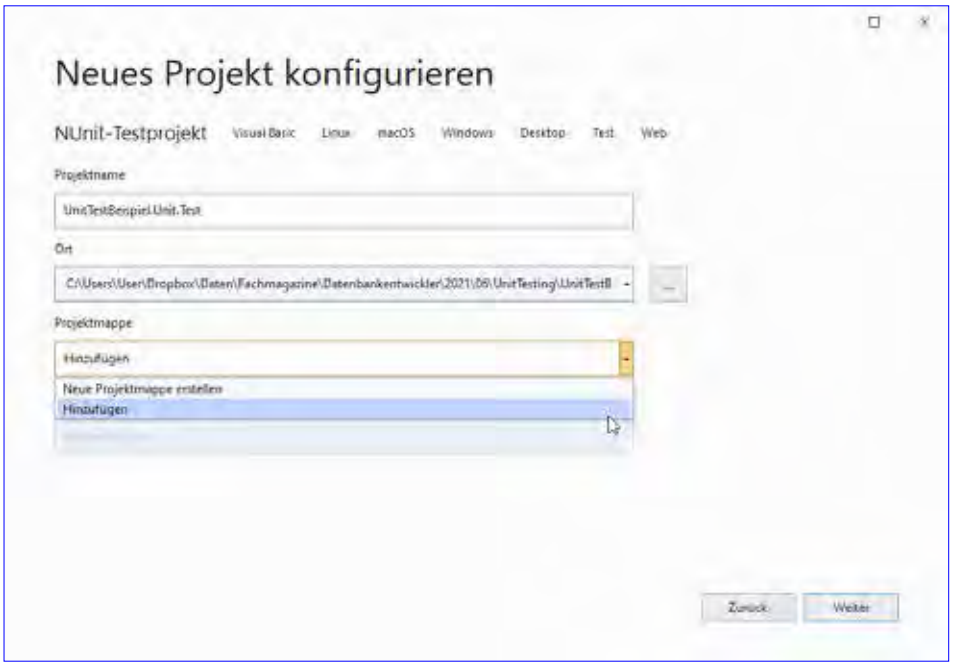


Bild 3: Hinzufügen des Projekts zur bestehenden Projektmappe

- Im folgenden Schritt legen Sie den neuen Projektnamen fest. Diesen wählen wir so, dass er mit dem Namen des zu testenden Projekts beginnt und hängen die Zeichenfolge **.Unit.Test** an. Außerdem stellen wir hier ein, dass das neue Projekt der bestehenden Projektemappe hinzugefügt werden soll (siehe Bild 3).

- Schließlich legen Sie noch das Zielframework für das neue Projekt fest.

Das gleiche Ergebnis erhalten Sie übrigens auch, wenn Sie den Menübefehl **Datei|Hinzufügen|Neues Projekt...** betätigen – Sie brauchen dann im Dialog **Neues Projekt konfigurieren** nicht mehr festzulegen, dass das neue Projekt zur aktuellen Projektmappe hinzugefügt werden soll.

Das neue Unit-Test-Projekt

Das Anlegen des Projekts hat zwei sichtbare Änderungen in unserem Projekt bewirkt. Die erste ist, dass wir im Projektmappen-Explorer unter unserem eigentlichen Projekt auch das neu angelegte Unit-Test-Projekt vorfinden (siehe Bild 4).

Die zweite Änderung ist das Klassenmodul **UnitTest1.vb**, das wir im Codefenster von Visual Studio vorfinden (siehe Bild 5). Die Bestandteile schauen wir uns gleich genauer an.

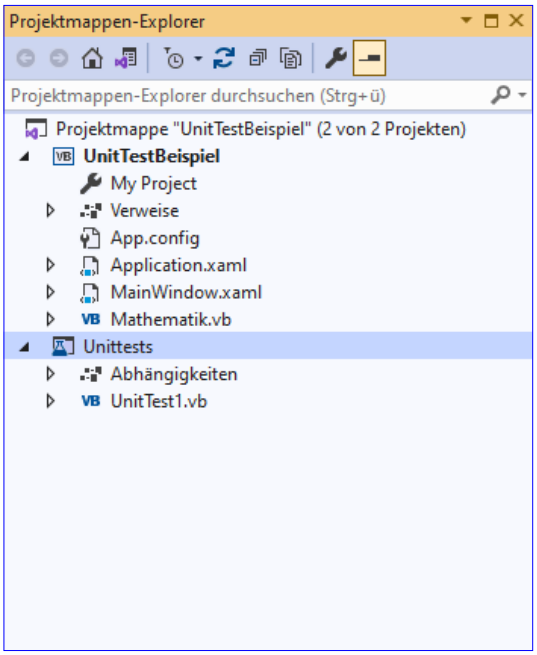


Bild 4: Das neue Testprojekt in der Projektmappe des zu testenden Projekts

Verweis zum Unit-Test-Projekt hinzufügen

Bevor wir uns die Unit-Test-Vorlage ansehen, die zum Testprojekt hinzugefügt wurde, wollen wir noch einen Verweis zum Testprojekt hinzufügen.

Dieser soll auf das zu testende Projekt verweisen. Um das zu erledigen, klicken Sie im Projektmappen-Explorer unter dem Testprojekt mit der rechten Maustaste auf den Eintrag **Abhängigkeiten** und wählen den Kontextmenübefehl **Projektverweis hinzufügen...** aus (siehe Bild 6).

Der nun erscheinende Verweis-Manager zeigt direkt die Projektmappe des zu testenden Projekts an. Hier setzen Sie einen Haken vor den Eintrag **UnitTestBeispiel** und schließen den Dialog wieder (siehe Bild 7).

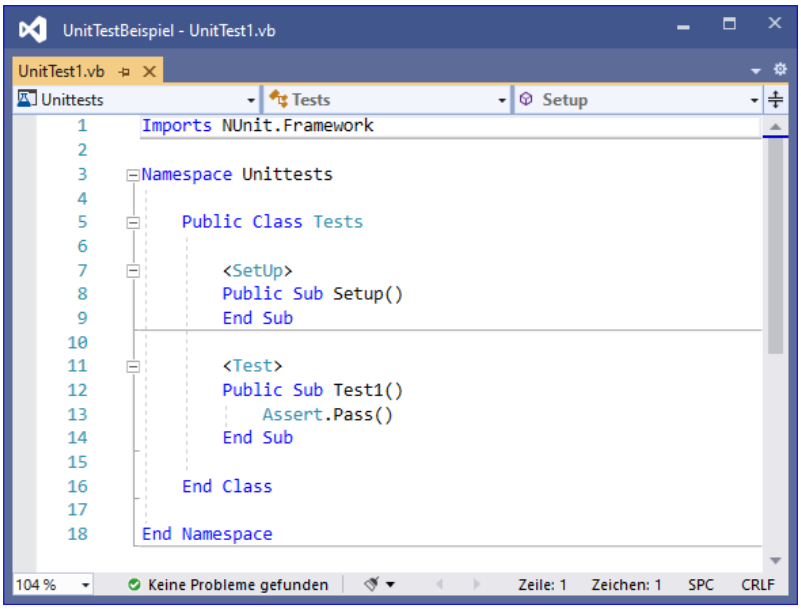


Bild 5: Die Vorlage für eine Testklasse

Die Unit-Test-Vorlage

Die unter dem Namen **UnitTest1.vb** hinzugefügte Klasse importiert zunächst den Namespace **NUnit.Framework**:

```
Imports NUnit.Framework
```

Dann deklariert sie einen Namespace namens **Unittests**, der eine erste Testklasse namens **Tests** enthält:

```
Namespace Unittests
    Public Class Tests
```

Hier finden wir einen Bereich, der mit dem Attribut **SetUp** versehen ist und eine Methode namens **Setup** enthält.

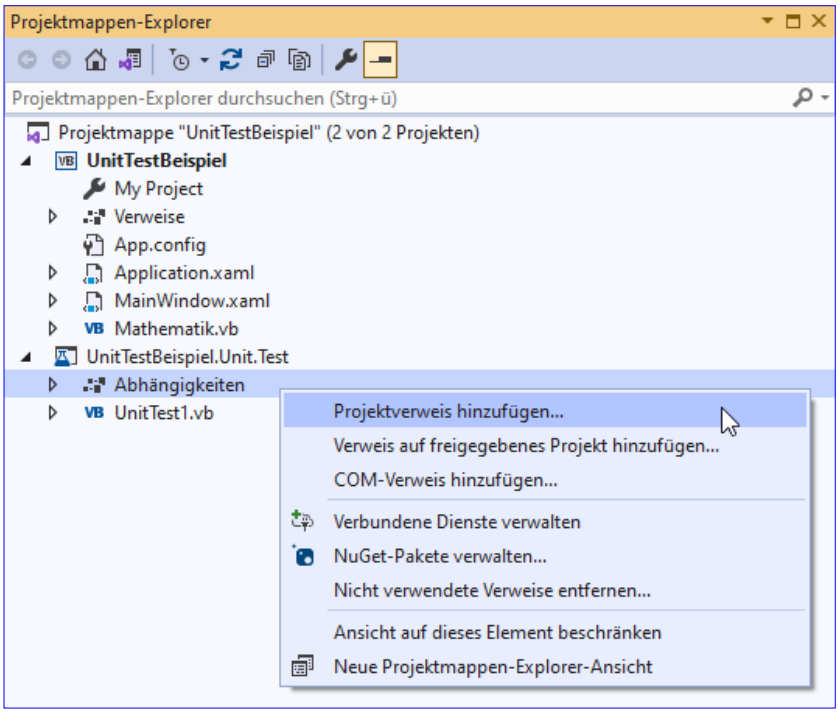


Bild 6: Hinzufügen eines Projektverweises

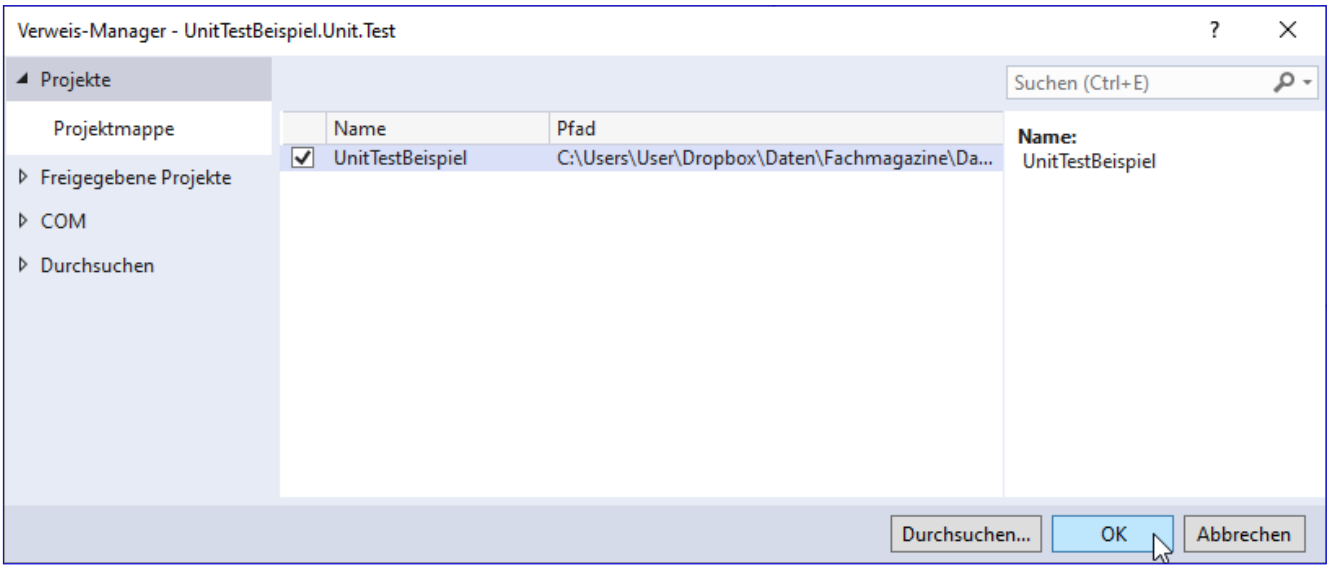


Bild 7: Hinzufügen des Verweises auf das zu testende Projekt

Dieser Methode können Sie Anweisungen hinzufügen, welche vor dem Aufruf einer jeden Testmethode ausgeführt werden:

```
        <SetUp>
        Public Sub Setup()
        End Sub
```

Danach folgen die eigentlichen Testmethoden. Diese werden mit dem Attribut **Test** versehen und sehen wie herkömmliche Methoden aus. Sie benötigen allerdings den Aufruf einer Methode der **Assert**-Klasse. Im Beispiel der Testvorlage lautet diese Methode **Pass**. **Pass** heißt, dass der Test bestanden wurde:

```
        <Test>
        Public Sub Test1()
            Assert.Pass()
        End Sub
    End Class
End Namespace
```

Den ersten Unit-Test durchführen

Um diesen Unit-Test zu starten, nutzen Sie die dafür vorgesehene Benutzeroberfläche.

Diese erscheint, wenn Sie den Menübefehl **Test|Alle Tests ausführen** aufrufen, automatisch (siehe Bild 8).

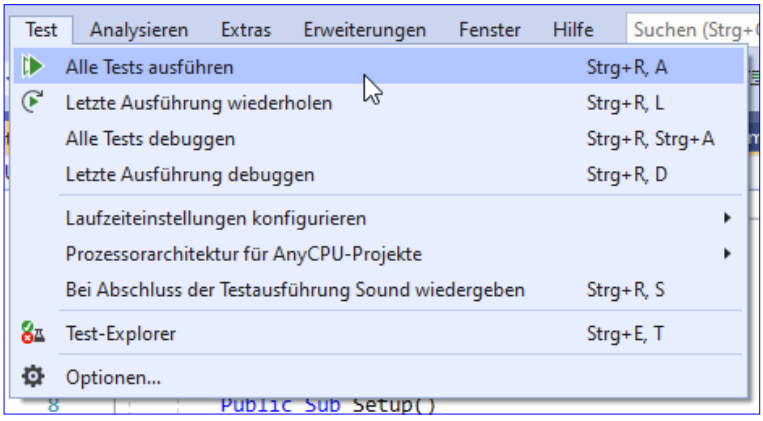


Bild 8: Starten der Tests

Seminarverwaltung IVa: Kunden und Seminare

Die Arbeit mit der Seminarverwaltung startet an dem Punkt, wo wir die online eingegangenen Bestellungen einlesen und in der Seminarverwaltung Kunden und Bestellungen in Kunden und Seminarteilnahmen umwandeln. Dazu benötigen wir eine Möglichkeit, die online erfolgten Bestellungen in die Anwendung einzulesen – siehe Artikel »Seminarverwaltung VI: Daten vom Shopsystem«. Außerdem wollen wir in diesem Artikel zeigen, wie Sie die Kunden und die Seminarteilnahmen verwalten können. Hier kann es auch vorkommen, dass ein Kunde auf einem anderen Wege als im Onlineshop bestellt – daher benötigen wir Möglichkeit zum Anlegen von Kunden und Seminarteilnahmen über die Benutzeroberfläche.

Wir haben diesen Artikel gegenüber der ursprünglichen Planung dreigeteilt. Den ersten Teil finden Sie hier vor, den zweiten und dritten Teil beschreiben wir gleich im Anschluss. In diesem Teil der Artikelreihe zeigen wir, wie Sie die Seiten zum Verwalten von Kunden und Seminaren und der Zuordnung von Kunden zu Seminaren und umgekehrt realisieren können. Dazu legen wir jeweils **Page**-Elemente an, und zwar die folgenden:

- **KundenUebersicht.xaml**: Diese Seite zeigt eine Liste aller Kunden an und bietet die Möglichkeit, die Kunden nach verschiedenen Kriterien zu durchsuchen. Außerdem enthält sie Schaltflächen zum Anlegen neuer Kunden, zum Löschen von Kunden und zum Bearbeiten eines Kunden.

- **Kundendetails.xaml**: Diese Seite zeigt die Details zu einem Kunden an. Dazu gehören neben den Daten der Tabelle **tblKunden** auch die von diesem Kunden gebuchten Seminare. Die Seite soll auch die Möglichkeit bieten, Seminare zum Kunden hinzuzufügen oder zu entfernen.

In der ersten Fortsetzung dieses Artikels namens **Seminarverwaltung IVb: Kunden und Seminare** (**www.datenbankentwickler.net/294**) schauen wir uns die Seiten zur Darstellung der Seminare in der Übersicht und in der Einzelansicht an. Die dort beschriebenen Seiten lauten:

- **SeminareUebersicht.xaml**: Die Übersichtsseite für die Seminare soll eine Liste aller Seminare anzeigen. Hier soll der Benutzer eine Suchfunktion vorfinden sowie Schaltflächen zum Anlegen neuer Seminare und zum Bearbeiten oder Löschen von Seminaren.

- **Seminardetails.xaml**: Die Detailseite eines Seminars zeigt neben den Details des Seminars auch eine Liste der Kunden an, die dieses Seminar gebucht haben. Sie können hier manuell Kunden hinzufügen oder aus der Liste der Teilnehmer entfernen. Außerdem finden Sie hier Funktionen, um den Kunden Informationen zum Seminar zuzusenden – beispielsweise E-Mails mit dem Link zur Teams-Sitzung, in der das Seminar stattfindet, oder für die Aufzeichnung des Seminars.

Schließlich folgt noch der dritte Teil, den wir **Seminarverwaltung IVc: Kunden und Seminare** (**www.datenbankentwickler.net/295**) genannt haben. In diesem beschreiben wir, wie wir den in den ersten beiden Teilen

erläuterten Seiten noch die Elemente hinzufügen, mit denen Sie die Zuordnung von Kunden zu Seminaren verwalten können.

Die Kundenübersicht

Die Kundenübersicht soll alle Kunden in einem **ListView**-Steuerelement anzeigen. Außerdem wollen wir folgende Funktionen bereitstellen:

- Öffnen der Details zu einem Kunden per Doppelklick auf den jeweiligen Eintrag
- Öffnen der Details zu einem Kunden durch Auswahl und anschließendes Betätigen der **Bearbeiten**-Schaltfläche
- Löschen des aktuell markierten Kunden per Schaltfläche
- Anlegen eines neuen Kunden per Schaltfläche
- Suche nach Kunden über den Vornamen und den Nachnamen

Der Entwurf dieser Seite namens **Kundendetails.xaml** sieht wie in Bild 1 aus.

Im XAML-Code haben wir ein Grid definiert, das drei Zeilen aufnimmt. Die erste enthält ein StackPanel mit den Suchfeldern, die zweite das **ListView**-Steuerelement und die dritte die Steuerelemente zum Verwalten der angezeigten Kunden:

```
<Page x:Class="KundenUebersicht" ... Title="KundenUebersicht">
    ...
    <Grid>
        <Grid.RowDefinitions>
            <RowDefinition Height="Auto"></RowDefinition>
            <RowDefinition Height="*"></RowDefinition>
```

Bild 1: Entwurf der Seite zur Anzeige der Kundenübersicht

```
        <RowDefinition Height="Auto"></RowDefinition>
    </Grid.RowDefinitions>
```

Das StackPanel für die Suche definieren wir mit zwei Textfeldern zur Eingabe von Vorname und/oder Nachname und einer Schaltfläche zum Ausführen der Suche:

```
    <StackPanel Orientation="Horizontal">
        <Label>Vorname:</Label>
        <TextBox x:Name="txtSucheVorname" Width="100"></TextBox>
        <Label>Nachname:</Label>
        <TextBox x:Name="txtSucheNachname" Width="100"></TextBox>
        <Button x:Name="btnSuchen" Click="btnSuchen_Click">
            <StackPanel Orientation="Horizontal">
                <Image Source="images/find_text.png" Width="24" Height="24"></Image>
            </StackPanel>
        </Button>
    </StackPanel>
```

Das **ListView**-Steuerelement namens **lvwKunden** ist an die Auflistung **Kunden** aus dem Code behind-Modul gebunden. Es erlaubt das Auswählen jeweils eines Elements und bei einem Doppelklick soll eine Ereignismethode ausgelöst werden. Die einzelnen anzuzeigenden Spalten definieren wir mit **GridViewColumn**-Elementen:

```
    <ListView x:Name="lvwKunden" Grid.Row="1" ItemsSource="{Binding Kunden}" SelectionMode="Single"
            MouseDoubleClick="lvwKunden_MouseDoubleClick">
        <ListView.View>
            <GridView>
                <GridViewColumn Header="ID" DisplayMemberBinding="{Binding ID}" />
                <GridViewColumn Header="Firma" DisplayMemberBinding="{Binding Firma}" />
                <GridViewColumn Header="Vorname" DisplayMemberBinding="{Binding Vorname}" />
                <GridViewColumn Header="Nachname" DisplayMemberBinding="{Binding Nachname}" />
                <GridViewColumn Header="Straße" DisplayMemberBinding="{Binding Strasse}" />
                <GridViewColumn Header="PLZ" DisplayMemberBinding="{Binding PLZ}" />
                <GridViewColumn Header="Ort" DisplayMemberBinding="{Binding Ort}" />
                <GridViewColumn Header="Land" DisplayMemberBinding="{Binding Land}" />
                <GridViewColumn Header="E-Mail" DisplayMemberBinding="{Binding EMail}" />
            </GridView>
        </ListView.View>
    </ListView>
```

Die dritte Zeile des **Grid**-Elements enthält ein weiteres **StackPanel**-Element mit den Steuerelementen zum Verwalten der Kunden – hier mit Schaltflächen zum Schließen des Bereichs (**btnOK**), zum Anlegen neuer Kunden

(**btnNeu**), zum Anzeigen des aktuellen Kunden (**btnDetails**) und zum Löschen des aktuell markierten Kunden (**btnLoeschen**):

```
        <StackPanel Orientation="Horizontal" Grid.Row="2">
            <Button x:Name="btnOK" Click="btnOK_Click">
                <StackPanel Orientation="Horizontal">
                    <Image Source="images/ok.png" Width="24" Height="24"></Image>
                    <Label>OK</Label>
                </StackPanel>
            </Button>
            <Button x:Name="btnNeu" Click="btnNeu_Click">
                <StackPanel Orientation="Horizontal">
                    <Image Source="images/add.png" Width="24" Height="24"></Image>
                    <Label>Neu</Label>
                </StackPanel>
            </Button>
            <Button x:Name="btnDetails" Click="btnDetails_Click">
                <StackPanel Orientation="Horizontal">
                    <Image Source="images/pencil.png" Width="24" Height="24"></Image>
                    <Label>Bearbeiten</Label>
                </StackPanel>
            </Button>
            <Button x:Name="btnLoeschen" Click="btnLoeschen_Click">
                <StackPanel Orientation="Horizontal">
                    <Image Source="images/delete.png" Width="24" Height="24"></Image>
                    <Label>Löschen</Label>
                </StackPanel>
            </Button>
        </StackPanel>
    </Grid>
</Page>
```

Anzeigen der Kunden im ListView-Steuerelement

Im Code behind-Modul benötigen wir den folgenden Namespace-Verweis:

```
Imports System.Collections.ObjectModel
```

Außerdem deklarieren wir hier die folgenden Variablen:

```
Private dbContext As New SeminarverwaltungContext
Private _kunden As ObservableCollection(Of Kunde)
```

```
Private objFrame As Frame
Private objListCollectionView As ListCollectionView
```

Beim Initialisieren der Seite wird die Konstruktor-Methode ausgelöst. Diese erwartet als Parameter einen Verweis auf das übergeordnete **Frame**-Element, beispielsweise zum Leeren des **Frame**-Elements, wenn die Seite geschlossen wird. Diesen Verweis speichert die Prozedur in der Variablen **objFrame**. Außerdem initialisiert die Methode die Komponente (**InitializeComponent**), füllt die Liste der Kunden, weist sich selbst dem **DataContext** zu und referenziert mit **objListCollectionView** eine Sicht auf die Auflistung **Kunden**. Schließlich stellt sie die Methode **Kundenfilter** als auszuführende Methode beim Aufruf des Filters der **ListCollectionView** ein:

```
Public Sub New(fra As Frame)
    objFrame = fra
    InitializeComponent()
    Kunden = New ObservableCollection(Of Kunde)(dbContext.Kunden)
    DataContext = Me
    objListCollectionView = CollectionViewSource.GetDefaultView(Kunden)
    objListCollectionView.Filter = AddressOf Kundenfilter
End Sub
```

Die öffentliche Eigenschaft **Kunden** erfasst die Auflistung der Kunden und stellt diese als Quelle für das **ListView**-Steuerelement zur Verfügung:

```
Public Property Kunden As ObservableCollection(Of Kunde)
    Get
        Return _kunden
    End Get
    Set(value As ObservableCollection(Of Kunde))
        _kunden = value
    End Set
End Property
```

Filtern nach dem Vornamen und dem Nachnamen

Die Steuerelemente im oberen Bereich der Kundenübersicht erlauben die Eingabe eines Vornamens und einen Nachnamens, nach dem gefiltert werden soll. Für die Eigenschaft **Filter** des **ListCollectionView**-Elements haben wir weiter oben bereits die Funktion **Kundenfilter** zugewiesen. Die Funktion **Kundenfilter** wird beim Filtern der **ListCollectionView** für jeden Datensatz einmal aufgerufen. Dabei erfasst sie den Vornamen und den Nachnamen des aktuellen **Kunde**-Elements und schreibt diese beziehungsweise gegebenenfalls eine leere Zeichenkette in die Variablen **strVorname** und **strNachname**. Dann prüft sie, ob Nachname und Vorname mit den Werten aus **txtSucheNachname** beziehungsweise **txtSucheVorname** beginnen. Falls nicht, wird der Wert **False** zurückgegeben. Ist beides erfüllt, gibt die Funktion den Wert **True** zurück und des jeweilige Element wird angezeigt:

```
Private Function Kundenfilter(item As Object) As Boolean
    Dim objKunde As Kunde
    Dim strVorname As String
    Dim strNachname As String
    objKunde = TryCast(item, Kunde)
    strVorname = IIf(objKunde.Vorname = Nothing, "", objKunde.Vorname)
    strNachname = IIf(objKunde.Nachname = Nothing, "", objKunde.Nachname)
    If Not (strNachname.StartsWith(txtSucheNachname.Text, StringComparison.OrdinalIgnoreCase)) Then
        Return False
    End If
    If Not (strVorname.StartsWith(txtSucheVorname.Text, StringComparison.OrdinalIgnoreCase)) Then
        Return False
    End If
    Return True
End Function
```

Damit der Filter angewendet wird, muss der Benutzer noch die Schaltfläche **btnSuchen** betätigen. Diese ruft die Methode **Refresh** der **ListCollectionView** auf, wodurch der Filter neu gesetzt wird:

```
Private Sub btnSuchen_Click(sender As Object, e As RoutedEventArgs)
    objListCollectionView.Refresh()
End Sub
```

Einen Eintrag löschen

Die Schaltfläche **btnLoeschen** soll den aktuell markierten Eintrag löschen. Die dadurch ausgelöste Methode sieht wie folgt aus:

```
Private Sub btnLoeschen_Click(sender As Object, e As RoutedEventArgs)
    Dim ZuLoeschenderKunde As Kunde
    ZuLoeschenderKunde = lvwKunden.SelectedItem
    If Not ZuLoeschenderKunde Is Nothing Then
        With dbContext
            .Kunden.Remove(ZuLoeschenderKunde)
            .SaveChanges()
        End With
        Kunden.Remove(ZuLoeschenderKunde)
    Else
            MessageBox.Show("Markieren Sie den zu löschenden Eintrag.", "Kein Eintrag markiert", _
                MessageBoxButton.OK, MessageBoxImage.Information)
    End If
End Sub
```

Die Methode ermittelt zunächst über die Eigenschaft **SelectedItem** das aktuell markierte Element im **ListView**-Steuerelement und referenziert es mit der Variablen **ZuLoeschenderKunde**. Dann prüft sie, ob überhaupt ein Eintrag markiert ist.

Falls ja, entfernt die Methode diesen zunächst aus der Auflistung **Kunden** des **dbContext**-Elements und überträgt die Änderung mit der **SaveChanges**-Methode in die zugrunde liegende Datenbank. Anschließend entfernt sie das Element auch noch aus der Auflistung **Kunden** und somit auch aus dem **ListView**-Steuerelement.

Hat der Benutzer zu diesem Zeitpunkt keinen Eintrag im **ListView**-Steuerelement markiert, erscheint eine entsprechende Meldung.

Details zu einem Kunden anzeigen

Betätigt der Benutzer die Schaltfläche **btnDetails** oder klickt er doppelt auf einen der Einträge im **ListView**-Steuerelement, soll die Seite **Kundendetails.xaml** die Details zum aktuellen Eintrag der Liste anzeigen. Beim Betätigen der Schaltfläche **btnDetails** prüft die folgende Methode, ob ein Kunde markiert ist, bevor sie die Methode **KundendetailsAnzeigen** aufruft. Anderenfalls folgt ein entsprechender Hinweis per Meldung:

```
Private Sub btnDetails_Click(sender As Object, e As RoutedEventArgs)
    If Not lvwKunden.SelectedItem Is Nothing Then
        KundendetailsAnzeigen()
    Else
        MessageBox.Show("Markieren Sie den zu anzuzeigenden Eintrag.", "Kein Eintrag markiert", _
            MessageBoxButton.OK, MessageBoxImage.Information)
    End If
End Sub
```

Beim Doppelklick auf einen der Listeneinträge ruft die folgende Ereignismethode die Routine **KundendetailsAnzeigen** auf:

```
Private Sub lvwKunden_MouseDoubleClick(sender As Object, e As MouseButtonEventArgs)
    KundendetailsAnzeigen
End Sub
```

Die Methode zum Anzeigen des aktuell markierten Kunden sieht nun wie folgt aus:

```
Private Sub KundendetailsAnzeigen()
    Dim pgeKundendetails As Kundendetails
    Dim MarkierterKunde As Kunde
    MarkierterKunde = lvwKunden.SelectedItem
    pgeKundendetails = New Kundendetails(objFrame, dbContext, MarkierterKunde)
    AddHandler pgeKundendetails.KundeChanged, AddressOf Kundendetails_KundeChanged
```

Seminarverwaltung IVb: Kunden und Seminare

In diesem Teil der Artikelreihe zur Seminarverwaltung zeigen wir, wie Sie die Seminare verwalten. Dazu benötigen wir zwei Seiten – eine zur Anzeige der Übersicht aller Seminare und eine zur Anzeige der Details eines einzelnen Seminars beziehungsweise zum Anlegen eines neuen Seminars. Dabei müssen wir noch eine kleine Anpassung am Entity Date Model durchführen, und außerdem legen wir noch die Methoden im Hauptfenster an, mit denen Sie die beiden Seiten zur Anzeige und Bearbeitung der Seminare darstellen

Wir haben diesen Artikel gegenüber der ursprünglichen Planung dreigeteilt. Den zweiten Teil finden Sie hier vor. Im ersten Teil namens **Seminarverwaltung IVa: Kunden und Seminare** (**www.datenbankentwickler.net/273**) stellen wir die beiden Seiten zur Anzeige der Übersicht und der Details der Kunden vor.

Im dritten Teil zeigen wir unter dem Titel **Seminarverwaltung IVc: Kunden und Seminare** (**www.datenbankentwickler.net/295**), wie Sie den Kunden Seminare zuordnen und umgekehrt.

Im aktuellen Teil der Artikelreihe schauen wir uns die Seiten zur Darstellung der Seminare in der Übersicht und in der Einzelansicht an. Die hier beschriebenen Seiten lauten:

- **SeminareUebersicht.xaml**: Die Übersichtsseite für die Seminare soll eine Liste aller Seminare anzeigen. Hier soll der Benutzer eine Suchfunktion vorfinden sowie Schaltflächen zum Anlegen neuer Seminare und zum Bearbeiten oder Löschen von Seminaren.

- **Seminardetails.xaml**: Die Detailseite eines Seminars zeigt neben den Details des Seminars auch eine Liste der Kunden an, die dieses Seminar gebucht haben. Sie können hier manuell Kunden hinzufügen oder aus der Liste der Teilnehmer entfernen. Außerdem finden Sie hier Funktionen, um den Kunden Informationen zum Seminar zuzusenden – beispielsweise E-Mails mit dem Link zur Teams-Sitzung, in der das Seminar stattfindet, oder für die Aufzeichnung des Seminars.

Änderungen am Entity Data Model

Die Entitäten für die Seminare enthalten ein Datumsfeld. Damit dieses beim Speichern in der Datenbank korrekt verarbeitet werden kann, fügen wir der Methode **OnModelCreating** eine Anweisung hinzu, welche den Datentyp für die Eigenschaft **Seminartermin** auf **datetime2** festlegt:

```
Protected Overrides Sub OnModelCreating(modelBuilder As DbModelBuilder)
    MyBase.OnModelCreating(modelBuilder)
    ...
    modelBuilder.Entity(Of Seminar).
        Property(Function(s) s.Seminartermin).HasColumnType("datetime2").HasPrecision(0)
End Sub
```

Aufruf der Seminarübersicht über das Ribbon

Im Artikel **Seminarverwaltung II: Ribbon und Frame** (**www.datenbankentwickler.net/272**) haben wir bereits den Aufbau des Ribbons für diese Anwendung beschrieben. Was noch fehlt, sind die Ereignismethoden, die durch die Schaltflächen **btnSeminaruebersicht** und **btnSeminaranlegen** ausgelöst werden. Mit diesen Schaltflächen wollen wir die noch zu erstellenden Seiten **SeminareUebersicht** und **Seminardetails** aufrufen. Die Methode, die durch die Schaltfläche **btnSeminaruebersicht** ausgelöst werden soll, erstellt eine neue Seite auf Basis des **Page**-Elements **SeminareUebersicht** und übergibt dieser einen Verweis auf das **Frame**-Objekt und auf den Datenbankkontext aus **dbContext**. Dann weist sie der Eigenschaft **Content** des **Frame**-Elements das neue **Page**-Element zu:

```
Private Sub btnSeminaruebersicht_Click(sender As Object, e As RoutedEventArgs)
    Dim pgeSeminaruebersicht As New SeminareUebersicht(fra, dbContext)
    fra.Content = pgeSeminaruebersicht
End Sub
```

Aufruf der Seite zum Anlegen eines neuen Seminars über das Ribbon

Ähnlich sieht der Aufruf für die Schaltfläche **btnSeminarAnlegen** aus:

```
Private Sub btnSeminarAnlegen_Click(sender As Object, e As RoutedEventArgs)
    Dim pgeSeminardetails As New Seminardetails(fra, dbContext)
    fra.Content = pgeSeminardetails
End Sub
```

Validierung der Seminare bei der Eingabe

Bevor wir die Benutzeroberfläche und die Anwendungslogik für das Anlegen neuer Seminare und das Bearbeiten vorhandener Seminare kreieren, benötigen wir noch eine Klasse, welche die Validierungsregeln für die **Seminar**-Elemente enthält. Dafür legen wir eine Klasse namens **Seminar.vb** an, in der wir die Schnittstelle **IDataErrorInfo** implementieren. Diese enthält die beiden Eigenschaften **Item** und **Error**. Für die Eigenschaft **Item** hinterlegen wir die Validierungsregeln, die Eigenschaft Error brauchen wir nicht anzupassen.

Die beiden Felder **Titel** und **Inhalt** dürfen einfach nicht leer sein, was wir durch die Prüfung des Inhalts dieser Felder mit **String.IsNullOrEmpty** prüfen. Der Inhalt des Feldes **Preis** darf nicht leer sein und weder einen negativen noch einen nicht-numerischen Wert enthalten. Das Feld **Seminartermin** darf ebenfalls nicht leer sein und muss ein Datum enthalten:

```
Imports System.ComponentModel

Public Class Seminar
    Implements IDataErrorInfo
    Default Public ReadOnly Property Item(columnName As String) As String Implements IDataErrorInfo.Item
        Get
```

```
            Dim strErrorMessage As String = ""
            Select Case columnName
                Case "Titel"
                    If (String.IsNullOrEmpty(Titel)) Then
                        strErrorMessage = "Bitte geben Sie einen Titel ein."
                    End If
                Case "Inhalt"
                    If (String.IsNullOrEmpty(Inhalt)) Then
                        strErrorMessage = "Bitte geben Sie einen Inhalt ein."
                    End If
                Case "Preis"
                    If (Preis < 0 Or String.IsNullOrEmpty(Preis)) Then
                        strErrorMessage = "Bitte geben Sie einen Preis ein."
                    End If
                Case "Seminartermin"
                    If (String.IsNullOrEmpty(Seminartermin) Or Not IsDate(Seminartermin)) Then
                        strErrorMessage = "Bitte geben Sie einen Seminartermin ein."
                    End If
            End Select
            Return strErrorMessage
        End Get
    End Property

    Public ReadOnly Property [Error] As String Implements IDataErrorInfo.Error
        Get
            'Throw New NotImplementedException()
        End Get
    End Property
End Class
```

Detailseite zum Anlegen und Bearbeiten von Seminaren

Damit gehen wir gleich über zu der Seite, mit der wir neue Seminare anlegen und vorhandene Seminare bearbeiten wollen. Im Teil mit den **Resources** für das **Page**-Element definieren wir wieder einige allgemeine Eigenschaften für die Steuerelemente wie **Margin** und **Padding** für die Elemente **DatePicker** und **Button**. Für das Element **TextBox** legen wir außerdem noch einen Wert für das Attribut **Validation.ErrorTemplate** fest, mit dem wir das Aussehen der **TextBox**-Elemente für den Fall definieren, dass der Inhalt nicht erfolgreich validiert werden konnte. Diese Teile finden Sie im Beispielprojekt, wir haben sie an dieser Stelle aus Platzgründen ausgespart:

```
<Page x:Class="Seminardetails"... Title="Seminardetails">
    <Page.Resources>
        ...
```

Ein weiteres **Style**-Element definiert die Regeln, nach denen **Button**-Elemente, die diese Regel aufnehmen, deaktiviert werden sollen. Diese Regeln richten sich nach der Validierung, genauer danach, ob die Validierung für ein bestimmtes Steuerelement fehlgeschlagen ist.

```
    <Style x:Key="EnableOnValidation" TargetType="{x:Type Button}">
        <Style.Triggers>
            <DataTrigger Binding="{Binding ElementName=txtTitel, Path=(Validation.HasError)}" Value="True">
                <Setter Property="IsEnabled" Value="False"></Setter>
            </DataTrigger>
            <DataTrigger Binding="{Binding ElementName=txtInhalt, Path=(Validation.HasError)}" Value="True">
                <Setter Property="IsEnabled" Value="False"></Setter>
            </DataTrigger>
            <DataTrigger Binding="{Binding ElementName=txtPreis, Path=(Validation.HasError)}" Value="True">
                <Setter Property="IsEnabled" Value="False"></Setter>
            </DataTrigger>
            <DataTrigger Binding="{Binding ElementName=txtSeminartermin, Path=(Validation.HasError)}"
                    Value="True">
                <Setter Property="IsEnabled" Value="False"></Setter>
            </DataTrigger>
        </Style.Triggers>
    </Style>
</Page.Resources>
```

Anschließend folgt die Definition für die **RowDefinition**- und die **ColumnDefinition**-Elemente für das **Grid** dieses **Page**-Elements, die wir hier ebenfalls nicht abbilden.

Steuerelemente der Seite Seminardetails.xaml

Interessanter wird es wieder bei der Definition der Steuerelemente der Seite **Seminardetails.xaml**, die wie in Bild 1 aussehen sollen. Hier finden wir zwei Spalten, von denen die erste jeweils das **Label**-Element aufnimmt und die zweite die eigentlichen gebundenen Steuerelemente.

Damit die Steuerelemente die Werte des jeweiligen Datensatzes beziehungsweise des jeweiligen Elements des Entity Data Models anzeigen, werden wir im Anschluss das Code behind-Modul mit einer Eigenschaft

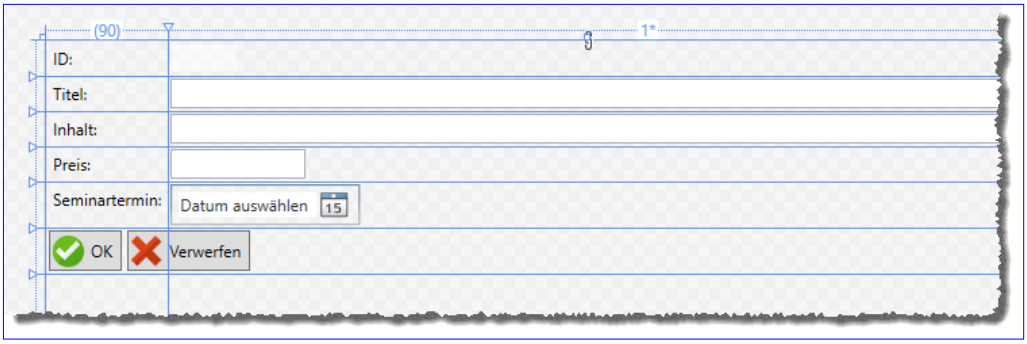


Bild 1: Entwurf der Seite zur Anzeige der Seminardetails

namens **AktuellesSeminar** füllen, welches wir beim Anlegen eines neuen Seminars oder beim Bearbeiten eines vorhandenen Seminars jeweils mit dem entsprechenden Element füllen.

Für die zu bindende Eigenschaft geben wir daher einen Ausdruck wie **{Binding AktuellesSeminar.ID, Mode=TwoWay}** an – hier für das Binden an das Feld **ID**. Für die Elemente mit Validierung definieren wir zusätzlich noch das Attribut **ValidatesOnDataErrors** wie in **{Binding AktuellesSeminar.Titel, Mode=TwoWay, ValidatesOnDataErrors=True}**.

Die ID bilden wir in einem **TextBox**-Element ab, das wir mit dem Wert **False** für das Attribut **IsEnabled** für die Eingabe deaktivieren. Die Felder **Titel** und **Inhalt** werden jeweils mit **TextBox**-Elementen dargestellt, die über das Attribut **Text** an diese Felder gebunden sind.

Für das Feld **Preis** legen wir noch zur Anzeige des Wertes als Währung noch ein spezielles Format fest, nämlich **StringFormat**: **{Binding AktuellesSeminar.Preis, Mode=TwoWay, ValidatesOnDataErrors=True, StringFormat=C}**.

Das Datum zeigen wir mit einem **DatePicker**-Steuerelement an, die im Feld **Seminartermin** enthaltene Uhrzeit liefern wir mit einer separaten TextBox, die wir ebenfalls an das Feld **Seminartermin** binden. Allerdings legen wir hier mit **StringFormat='t'** fest, dass nur die Uhrzeit angezeigt wird.

Die gebundenen Felder definieren wir insgesamt wie folgt:

```
<Grid>
    ...
    <Label Content="ID:" Grid.Column="0" />
    <TextBox x:Name="txtID" Grid.Column="1" HorizontalAlignment="Left" Text="{Binding
        AktuellesSeminar.ID, Mode=TwoWay}" Width="50" IsEnabled="False" BorderBrush="Transparent" />
    <Label Content="Titel:" Grid.Column="0" Grid.Row="1" />
    <TextBox x:Name="txtTitel" Grid.Column="1" Grid.Row="1" HorizontalAlignment="Stretch"
        Text="{Binding AktuellesSeminar.Titel, Mode=TwoWay, ValidatesOnDataErrors=True}" />
    <Label Content="Inhalt:" Grid.Row="2" />
    <TextBox x:Name="txtInhalt" Grid.Column="1" Grid.Row="2" HorizontalAlignment="Stretch"
        Text="{Binding AktuellesSeminar.Inhalt, Mode=TwoWay, ValidatesOnDataErrors=True}" />
    <Label Content="Preis:" Grid.Row="3" />
    <TextBox x:Name="txtPreis" Grid.Column="1" Grid.Row="3" Width="100" Text="{Binding
        AktuellesSeminar.Preis, Mode=TwoWay, ValidatesOnDataErrors=True, StringFormat=C}" />
    <Label Content="Seminartermin:" Grid.Row="4" />
    <DatePicker x:Name="dpSeminartermin" Grid.Column="1" Grid.Row="4" Width="140"
        HorizontalAlignment="left" SelectedDate="{Binding AktuellesSeminar.Seminartermin, Mode=TwoWay,
        ValidatesOnDataErrors=True, TargetNullValue=''}"></DatePicker>
   <Label Content="Uhrzeit:" Grid.Row="5" />
```

```
        <TextBox x:Name="txtUhrzeit" Grid.Column="1" Grid.Row="5" Width="140" HorizontalAlignment="left"
            Text="{Binding AktuellesSeminar.Seminartermin, Mode=TwoWay, ValidatesOnDataErrors=True, Tar-
            getNullValue='', StringFormat='t'}"></TextBox>
```

Schaltflächen zum Speichern und zum Verwerfen eines Seminars

Damit fehlen noch die beiden Schaltflächen, die wir in einem **StackPanel**-Element nebeneinander anordnen. Die Schaltfläche **btnSpeichern** soll nur aktiviert werden, wenn die in der weiter oben definierten statischen Ressource **EnableOnValidation** gestellten Bedingungen erfüllt sind.

Deshalb legen wir im **Style**-Element der Schaltfläche mit dem **BasedOn** fest, das die dortigen Bedingungen für den Wert des Attributs **Enabled** geprüft werden sollen. Die beiden Schaltflächen enthalten wiederum jeweils ein **StackPanel**-Element, welches nebeneinander das Icon und den Schaltflächentext beinhaltet:

```
        <StackPanel Orientation="Horizontal" Grid.Row="6" Grid.ColumnSpan="4" >
            <Button x:Name="btnSpeichern" Click="btnSpeichern_Click">
                <Button.Style>
                    <Style TargetType="{x:Type Button}" BasedOn="{StaticResource EnableOnValidation}">
                        <Style.Setters>
                            <Setter Property="Margin" Value="2"></Setter>
                        </Style.Setters>
                    </Style>
                </Button.Style>
                <StackPanel Orientation="Horizontal">
                    <Image Source="images/ok.png" Width="24" Height="24"></Image>
                    <Label>OK</Label>
                </StackPanel>
            </Button>
            <Button x:Name="btnVerwerfen" Click="btnVerwerfen_Click">
                <StackPanel Orientation="Horizontal">
                    <Image Source="images/delete.png" Height="24" Width="24"></Image>
                    <Label>Verwerfen</Label>
                </StackPanel>
            </Button>
        </StackPanel>
    </Grid>
</Page>
```

Anwendungslogik in der Datei Seminarverwaltung.xaml.vb

Bevor wir die Seite testen können, benötigen wir noch die Anwendungslogik hinter der Seite. Dazu hinterlegen wir in der Datei **Seminarverwaltung.xaml.vb** zunächst zwei **Namespace**-Importe:

```
Imports System.Collections.ObjectModel
Imports System.Data.Entity
```

Die Klasse Seminardetails deklariert dann Variablen für den Datenbankkontext, das aktuell dargestellte Seminar, das umgebende **Frame**-Element und die Auflistung aller Seminare. Außerdem eine Variable namens **bolVonUebersicht**, die angibt, ob die Seite von der Übersichtsseite aus geöffnet wurde:

```
Class Seminardetails
    Private dbContext As SeminarverwaltungContext
    Public Property AktuellesSeminar As Seminar
    Private fra As Frame
    Private Seminare As ObservableCollection(Of Seminar)
    Private bolVonUebersicht As Boolean
```

Außerdem deklarieren wir ein öffentliches Ereignis, das ausgelöst wird, wenn das Seminar geändert wurde:

```
    Public Event SeminarChanged(ByVal sender As Object, ByVal e As EventArgs)
```

Konstruktoren von Seminardetails.xaml.vb

Die Seite Seminardetails erhält zwei Konstruktoren. Der erste erwartet neben dem Verweis auf das umschließende **Frame**-Objekt und dem Datenbankkontext gegebenenfalls noch ein **ObservableCollection**-Objekt mit den Seminaren der Anwendung. Dieser Konstruktor wird verwendet, wenn mit der Seite ein neues **Seminar**-Element angelegt werden soll. Die Konstruktormethode initialisiert die Seite und weist der Variablen **fra** das übergebene **Frame**-Objekt zu. Dann erstellt sie ein neues **Seminar**-Element und referenziert es mit der Variablen **AktuellesSeminar**. Die Auflistung **objSeminare** enthält bereits eine ObservableCollection, wenn die Seite von der Übersichtsseite aufgerufen wurde, die wir im Anschluss besprechen. Anderenfalls wird die **ObservableCollection** erstellt und mit den Elementen der Auflistung **Seminare** gefüllt. Das Element **AktuellesSeminar** wird der Auflistung **Seminare** hinzugefügt und auch der entsprechenden **DbSet**-Auflistung des Datenbankkontextes. Schließlich stellt der Konstruktor den **DataContext** der Seite auf das Code behind-Modul selbst ein:

```
    Public Sub New(objFrame As Frame, Optional dbC As SeminarverwaltungContext = Nothing, _
            Optional objSeminare As ObservableCollection(Of Seminar) = Nothing)
        InitializeComponent()
        fra = objFrame
        AktuellesSeminar = New Seminar
        dbContext = dbC
        If objSeminare Is Nothing Then
            Seminare = New ObservableCollection(Of Seminar)(dbContext.Seminare)
        Else
            Seminare = objSeminare
        End If
```

```
        Seminare.Add(AktuellesSeminar)
        dbContext.Seminare.Add(AktuellesSeminar)
        DataContext = Me
    End Sub
```

Die zweite Konstruktormethode wird aufgerufen, wenn ein bereits vorhandenes Seminar angezeigt werden soll. Dies geschieht von der Übersichtsseite aus, die wir später vorstellen. Die Methode erwartet ebenfalls einen Verweis auf das **Frame**-Objekt sowie den Datenbankkontext und außerdem einen Verweis auf das anzuzeigende **Seminar**-Element.

Sie referenziert den Datenbankkontext und das **Frame**-Objekt mit den entsprechenden Parameterwerten und stellt die Variable **AktuellesSeminar** auf das übergebene Seminar ein. Außerdem stellt die Methode die Variable **bolVonUebersicht** auf **True** ein, da es von der Übersichtsseite aus aufgerufen wurde:

```
    Public Sub New(objFrame As Frame, dbC As SeminarverwaltungContext, objSeminar As Seminar)
        InitializeComponent()
        dbContext = dbC
        fra = objFrame
        AktuellesSeminar = objSeminar
        DataContext = Me
        bolVonUebersicht = True
    End Sub
```

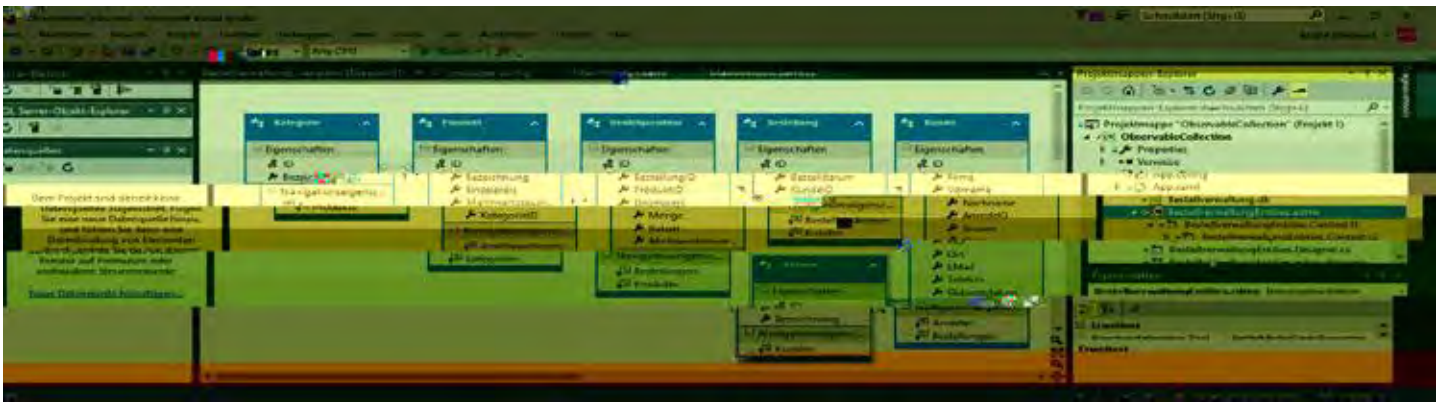
Schaltfläche zum Speichern des Seminars

Wenn Sie die Daten eines neuen Seminars eingegeben oder ein bestehendes Seminar angepasst haben, wollen Sie die Änderungen speichern. Das erledigen Sie mit der Schaltfläche **btnSpeichern**. Diese löst beim Anklicken die folgende Methode aus:

AUSGABE 01/2022

DATENBANK ENTWICKLER

MAGAZIN FÜR DIE DATENBANKENTWICKLUNG MIT VISUAL STUDIO FÜR DESKTOP, WEB UND CO.



TOP-THEMEN:

VB-BASICS	Bitmaps programmieren mit VB.NET	SEITE 3
VB-BASICS	Zeichnen in Bitmaps	SEITE 15
LÖSUNGEN	EPC-QR-Code für Rechnungen	SEITE 40
ENTWICKLUNG	Setup für COM-DLLs mit Inno Setup	SEITE 62

André Minhorst Verlag

Bitmaps programmieren mit VB.NET

Wer von Access/VBA kommt, kennt das Problem: Alles, was mit der Erstellung und Bearbeitung von Grafiken zu tun hat, ist sehr kompliziert und erfordert in der Regel die Verwendung von API-Funktionen. Unter VB.NET ist dies sehr viel einfacher. Daher bietet es sich nicht nur an, die Grafikfähigkeiten des Namespaces System.Drawing für .NET-Anwendungen zu nutzen. Sie können die gewünschten Funktionen beispielsweise auch in DLLs integrieren, die Sie dann von Access aus einbinden und aufrufen können. Im vorliegenden Artikel wollen wir uns jedoch zunächst einmal die grundlegenden Möglichkeiten zum Erstellen und Bearbeiten von Bilddateien unter .NET ansehen.

Vorbereitung: LINQPad

LINQPad ist unsere Spielwiese, wenn es um das Ausprobieren von VB.NET-Code geht. Hier brauchen wir zum Testen einiger Zeilen Code nicht extra ein VB-Projekt anzulegen und dieses jedes Mal neu zu starten, um die Funktionsweise zu prüfen. Wir geben die gewünschten Zeilen einfach in eine Methode ein und starten diese mit **F5** – ganz wie es beispielsweise im VBA-Editor möglich ist. Wie Sie LINQPad nutzen, lesen Sie im Artikel **LINQPad: LINQ, C# und VB einfach ausprobieren** (**www.datenbankentwickler.net/100**).

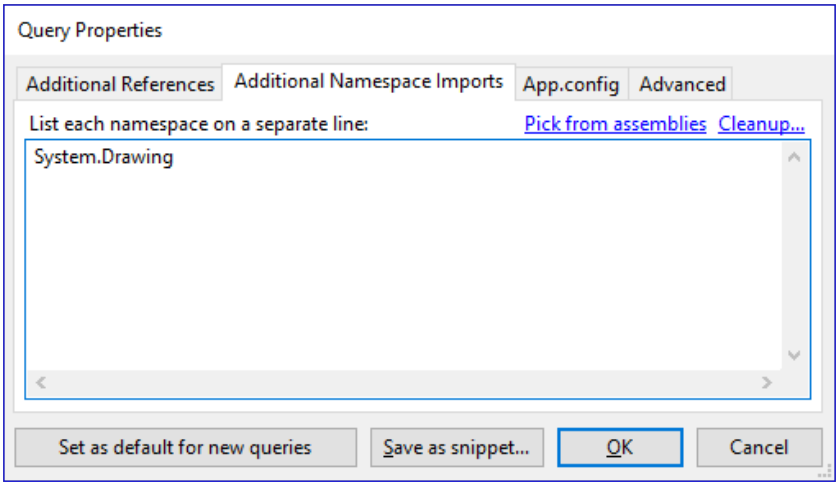


Bild 1: Importieren des Namespaces **System.Drawing**.

Für unser aktuelles Beispiel benötigen wir lediglich eine neue LINQPad-Query mit dem Wert **VB Program** im Auswahlfeld **Language**. Damit wir die Elemente des Namespaces **System.Drawing** verwenden können, fügen wir diesen zu der Query hinzu. Dazu klicken Sie mit der rechten Maustaste auf die Registerseite der Query und wählen den Eintrag **Query Properties...** aus. Hier wechseln Sie zur Registerseite **Additional Namespace Imports** und geben den Namespace **System.Drawing** einfach ein – ohne **Imports**-Anweisung, wie es direkt in Modulen nötig wäre (siehe Bild 1). Anschließend schließen Sie den Dialog mit der **OK**-Schaltfläche wieder.

Pfad zum Speichern der erstellten Bilddateien definieren

Die erzeugten Bilder wollen wir in einem Verzeichnis speichern, wo wir diese schnell wiederfinden. Den Namen dieses Verzeichnisses hinterlegen wir in einer Konstanten, die wir oben im Code wie folgt definieren:

```
Const strPath As String = "C:\...\BitmapTest\"
```

Eine neue Bilddatei erstellen

Wenn Sie einfach nur eine neue Bilddatei erstellen und unter einem bestimmten Namen speichern wollen, gelingt das sehr einfach. Dazu initialisieren Sie ein neues Objekt auf Basis der Klasse **Bitmap** und speichern diese einfach mit der **Save**-Methode. Diese erwartet den Namen der zu erstellenden Datei als Parameter. In einem einfachen Fall übergeben Sie die Höhe und die Breite der zu erstellenden Datei als Parameter beim Initialisieren des Objekts:

```
Dim bmp As New Bitmap(100, 100)
bmp.Save(strPath & "test.bmp")
```

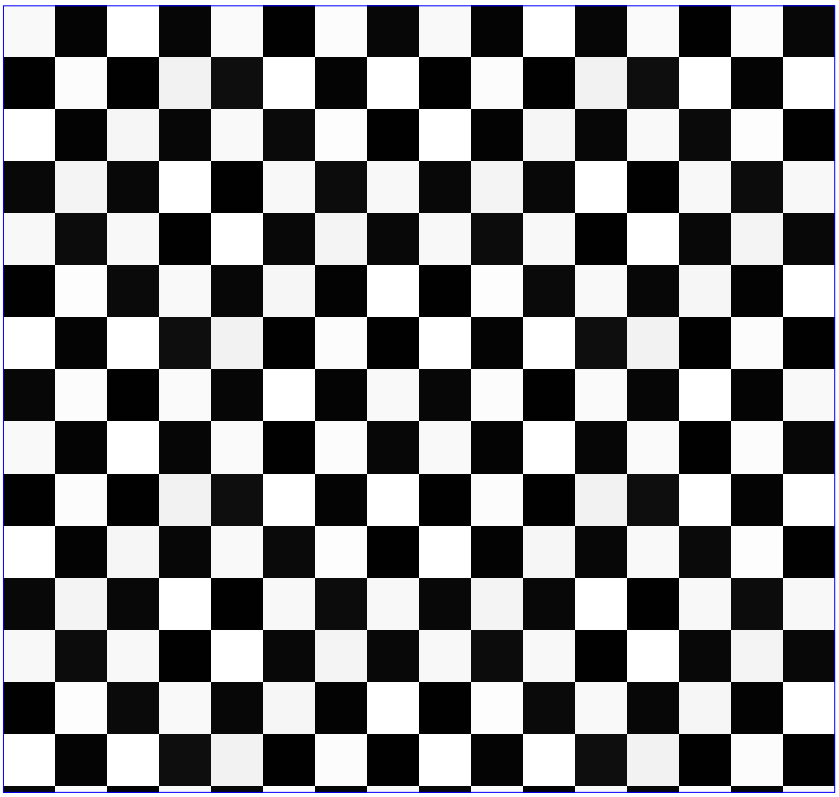


Bild 2: Eine frisch erstellte, komplett transparente Datei

Die so erzeugte Bitmap-Datei finden Sie im angegebenen Verzeichnis vor. Wenn Sie mit einem Bildprogramm öffnen, das in der Lage ist, transparente Bereiche hervorzuheben, sehen Sie, dass die erstellte Datei komplett transparent ist (siehe Bild 2).

Datei speichern und im Standardprogramm öffnen

Bevor wir uns weitere Techniken in Zusammenhang mit dem Bitmap-Objekt anschauen, wollen wir die Testprozedur noch ein wenig praktischer gestalten. Dazu fügen wir als Erstes eine neue Variable namens **strFilename** ein. Diese nimmt auch gleich den Namen der zu erstellenden Datei auf, hier **Test.bmp**. Der Grund ist, dass wir den Dateinamen gleich noch öfter benötigen. Wenn wir diesen dann ändern wollen, brauchen wir diesen nur an einer Stelle anzupassen.

Nach dem Erstellen und Speichern der Datei unter den angegebenen Dateinamen wollen wir diese noch in der dafür vorgesehenen Anwendung öffnen. Das erledigt ein Aufruf der **Start**-Methode der **Process**-Klasse:

```
Sub Main
    Dim bmp As New Bitmap(100, 100)
    Dim strFilename As String
    strFilename = "Test.bmp"
    bmp.Save(strPath & strFilename)
    Process.Start(strPath & strFilename)
End Sub
```

Hinweis: Wenn Sie die Standarddatei ändern wollen, die in Zusammenhang mit einer bestimmten Dateiendung aufgerufen wird, klicken Sie mit der rechten Maustaste auf den Dateinamen im Windows Explorer. Wählen Sie aus dem nun erscheinenden Kontextmenü den Eintrag **Öffnen mit|Andere App auswählen** aus. Selektieren Sie dort die gewünschte Anwendung und setzen Sie einen Haken für die Option **Immer diese App zum Öffnen von .bmp-Dateien verwenden** (siehe Bild 3).

Bild 3: Festlegen der Standardapp für eine Dateiendung

Bilddatei in verschiedenen Formaten speichern

Die **Save**-Methode bietet eine Überladung an, mit der Sie mit dem ersten Parameter den Namen der zu erstellenden Datei übergeben und mit dem zweiten das Bildformat. Hierfür können Sie die folgenden Werte angeben, die Sie über die Auflistung **Imaging.ImageFormat** finden:

- **Bmp**: Speichert das Bild im **Bitmap**-Format.
- **Emf**: Speichert das Bild im **EMF**-Format.
- **Gif**: Speichert das Bild im **GIF**-Format.
- **Jpeg**: Speichert das Bild im **JPEG**-Format.
- **Png**: Speichert das Bild im **PNG**-Format.
- **Tiff**: Speichert das Bild im **TIFF**-Format.
- **Wmf**: Speichert das Bild im **WMF**-Format.

Wenn wir also beispielsweise ein Bild im **PNG**-Format speichern wollen, nutzen wir den folgenden Aufruf der **Save**-Methode:

```
bmp.Save(strPfad & "test.png",Imaging.ImageFormat.Png)
```

Was ist ein Bitmap?

Ein Bitmap ist eine rechteckige Matrix aus wiederum viereckigen Elementen, die Pixel genannt werden. Für jedes Pixel wird ein Farbwert festgelegt. Eine Datei, die ein Bitmap enthält, besteht aus einigen Bytes mit grundlegenden Informationen über das Bitmap, die zum Beispiel die Anzahl der Pixel und die Anzahl der möglichen Farben je Pixel angeben, also die Farbtiefe.

Für die möglichen zur Verfügung stehenden Farben gibt es verschiedene Sets, vom einfachsten, das nur die beiden Farben Schwarz und Weiß enthält bis zu solchen, die 16.777.216 verschiedene Farben je Pixel darstellen können – und gegebenenfalls zusätzlich noch weitere 256 Abstufungen für einen sogenannten Alphakanal zum Definieren von Masken oder Transparenz.

Dementsprechend variiert die Speichergröße einer Datei mit einem Bitmap je nach der angegebenen Farbtiefe. Diese können wir übrigens über eine spezielle Eigenschaft ermitteln.

Wie aber erklären sich beispielsweise die 16.777.216 Farben für jedes Pixel? Diese resultieren aus dem Anteil der Farben Rot, Grün und Blau, die jeweils mit einem Wert von 0 bis 255 angegeben werden.

Es gibt auch noch Bitmaps mit Graustufen – beispielsweise mit 8 Bit. Das bedeutet, dass **0** einem schwarzen Pixel entspricht, **255** einem weißen und die Werte dazwischen geben Grautöne in den verschiedenen Helligkeiten wieder.

Farbtiefe und Farbraum eines Bitmap-Objekts ermitteln

Wenn wir wie oben ein einfaches **Bitmap**-Objekt mit einer beliebigen Auflösung erstellen, können wir mit der Eigenschaft **PixelFormat** eine Information über den Farbraum und die Farbtiefe ausgeben lassen:

```
Dim bmp As New Bitmap(100, 100)
Debug.Print (bmp.PixelFormat.ToString)
```

Die hier verwendete Enumeration **PixelFormat** finden wir im Namespace **System.Drawing.Imaging**, den wir wie **System.Drawing** zur Liste der Namespaces in der LINQPad-Abfrage hinzufügen.

Das Ergebnis lautet:

```
Format32bppArgb
```

Das bedeutet, dass wir ein **Bitmap**-Objekt mit 32-Bit-RGB erstellt haben, also ein 24-Bit-RGB-Bitmap mit 8-Bit-Alphakanal.

Farbtiefe und Farbraum eines Bitmap-Objekts einstellen

Sie können Farbtiefe und Farbraum auch direkt beim Erstellen eines **Bitmap**-Objekts definieren. Dazu nutzen Sie eine Überladung, bei der Sie als dritten Parameter einen Eintrag der Enumeration **PixelFormat** übergeben. Wenn Sie eine Alternative zum 32-Bit-Bitmap benötigen, finden Sie in der Enumeration ausreichend Möglichkeiten.

Da Sie meist auf Bilder mit 32-Bit stoßen werden, gehen wir an dieser Stelle nicht näher auf alternative Versionen ein.

Transparente Bilddatei erstellen

Gegebenenfalls stoßen Sie auf **Bitmap**-Objekte, beispielsweise aus Bilddateien, die keinen Alphakanal enthalten, aber Sie möchten diesen einen solchen hinzufügen. Wir demonstrieren das an einem mit dem Typ **Format24bppRgb** erstellten **Bitmap**-Objekt. Nachdem wir sicherheitshalber einmal den Wert der Eigenschaft **PixelFormat** ausgegeben haben, rufen wir die Methode **MakeTransparent** auf und übergeben dieser den Wert der Farbe, die transparent gemacht werden soll. Anschließend geben wir erneut den Typ aus der Eigenschaft **PixelFormat** aus:

```
Dim bmp As New Bitmap(16, 16, PixelFormat.Format24bppRgb)
Debug.Print(bmp.PixelFormat.ToString)
bmp.MakeTransparent(Color.Black)
Debug.Print(bmp.PixelFormat.ToString)
```

Das Ergebnis lautet:

```
Format24bppRgb
Format32bppArgb
```

Wir fügen mit der **MakeTransparent**-Methode also den Alphakanal zum **Bitmap**-Objekt hinzu.

Einem Pixel eine Farbe zuweisen

Haben Sie ein **Bitmap**-Objekt erstellt, möchten Sie den Pixeln des Objekts gegebenenfalls Farben zuweisen. Das erledigen Sie mit der Methode **SetPixel**. Diese erwartet zumindest die folgenden drei Parameter:

- **x**: x-Koordinate des Pixels
- **y**: y-Koordinate des Pixels
- **Color**: Farbe, die diesem Pixel zugefügt werden soll.

Die Werte für **x** und **y** sind **0**-basiert, bei einem **Bitmap**-Objekt mit 16x16 Pixeln erwartet die Methode **SetPixel** für die ersten beiden Parameter also Werte von **0** bis **15**.

Der dritte Parameter erwartet eine Struktur, welche die zu setzende Farbe repräsentiert. Hier können Sie durch Angabe der Klasse **Color** einige Einträge per IntelliSense auswählen, welche den gängigen Farbwerten entsprechen, beispielsweise **Color.White** oder **Color.Black**. Diese liefern **Color**-Objekte mit den Einstellungen für den **Rot**-, **Grün**- und **Blau**-Anteil sowie gegebenenfalls den Alphakanal und weiteren Eigenschaften.

Sie können die Farbe aber auch direkt aus den Werten für **R**, **G**, **B** und **A** zusammenstellen. Dazu nutzen Sie die Methode **FromArgb** der Klasse **Color** als dritten Parameter der **SetPixel**-Methode.

Zeichnen in Bitmaps mit der Graphics-Klasse

Im Artikel »Bitmaps programmieren mit VB.NET« haben wir gezeigt, wie Sie mit VB.NET Bitmaps erzeugen und die einzelnen Pixel mit Farben füllen beziehungsweise die enthaltenen Farben auslesen. Im vorliegenden Artikel gehen wir einen Schritt weiter: Wir nutzen eine Klasse namens Graphics, um in einem Bitmap zu zeichnen. Dabei zeigen wir die Techniken zum Erstellen der grundlegenden Formen und liefern Know-how, wie Sie Füllungen und Schraffuren hinzufügen. Schließlich speichern wir die erzeugten Bitmaps im Dateisystem und zeigen diese in der jeweiligen Standardapp für die entsprechende Dateiendung an.

Vorbereitung: LINQPad

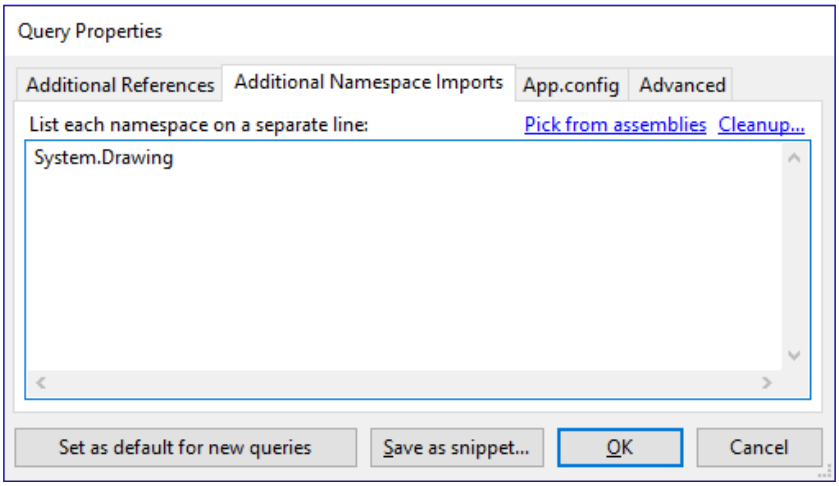


Bild 1: Hinzufügen eines Verweises auf den Namespace **System.Drawing**

Genau wie im oben genannten Artikel **Bitmaps programmieren mit VB.NET** (**www.datenbankentwickler.net/297**) verwenden wir auch in diesem Artikel LINQPad als Spielwiese zum Ausprobieren der Beispiele. Hier brauchen wir zum Testen einiger Zeilen Code nicht extra ein VB-Projekt anzulegen und dieses jedes Mal neu zu starten, um die Funktionsweise zu prüfen. Wir geben die gewünschten Zeilen einfach in eine Methode ein und starten diese mit **F5** – ganz wie es beispielsweise im VBA-Editor möglich ist. Wie Sie LINQPad nutzen, lesen Sie im Artikel **LINQPad: LINQ, C# und VB einfach ausprobieren** (**www.datenbankentwickler.net/100**).

Für unser aktuelles Beispiel benötigen wir lediglich eine neue LINQPad-Query mit dem Wert **VB Program** im Auswahlfeld **Language**. Damit wir die Elemente des Namespaces **System.Drawing** verwenden können, fügen wir diesen zu der Query hinzu. Dazu klicken Sie mit der rechten Maustaste auf den Registerreiter der Query und wählen den Eintrag **Query Properties...** aus. Hier wechseln Sie zur Registerseite **Additional Namespace Imports** und geben den Namespace **System.Drawing** einfach ein – ohne **Imports**-Anweisung, wie es direkt in Modulen nötig wäre (siehe Bild 1). Anschließend schließen Sie den Dialog mit der **OK**-Schaltfläche wieder.

Auch in diesem Artikel wollen wir die erzeugten Bilder in einem Verzeichnis speichern, wo wir diese schnell wiederfinden. Den Namen dieses Verzeichnisses hinterlegen wir in einer Konstanten, die wir oben im Code wie folgt definieren:

```
Const strPath As String = "C:\...\BitmapTest\"
```

Zeichnen mit Graphics

Während Sie in Objekten der **Bitmap**-Klasse mit der Methode **SetPixel** nur die Farbe für einzelne Pixel festlegen können, bietet die **Graphics**-Klasse einige weitere Möglichkeiten. Die **Graphics**-Klasse stellt Methoden wie **DrawLine** zum Zeichnen von Linien oder **DrawString** zum Einfügen von Texten zur Verfügung.

Dabei referenziert das Objekt auf Basis der **Graphics**-Klasse zunächst das Element, dem beispielsweise Linien oder Texte hinzugefügt werden sollen. Um die im oben genannten Artikel vorgestellte **Bitmap**-Klasse direkt einzubinden, nutzen wir die dort erzeugten Bitmaps quasi als Zeichenblatt. Das **Graphics**-Objekt füllen wir dabei mit der **FromImage**-Methode mit dem **Bitmap**-Element.

```
Dim bmp As New Bitmap(480, 320)
Dim objGraphics As Graphics
objGraphics = Graphics.FromImage(bmp)
```

Danach können wir die nachfolgenden Methoden nutzen, um dem **Bitmap**-Objekt über die **Graphics**-Klasse Elemente hinzuzufügen:

- **DrawArc**: Zeichnet einen Bogen.
- **DrawBezier**: Zeichnet eine Bézier-Kurve.
- **DrawBeziers**: Zeichnet mehrere Bézier-Kurven.
- **DrawClosedCurve**: Zeichnet eine geschlossene Cardinal-Splinekurve.
- **DrawEllipse**: Zeichnet eine Ellipse.
- **DrawIcon**: Zeichnet das durch das angegebene Icon dargestellte Bild.
- **DrawIconUnstretched**: Zeichnet das durch das angegebene Icon dargestellte Bild, ohne das Bild zu skalieren.
- **DrawImage**: Zeichnet das angegebene Bild in seiner ursprünglichen physischen Größe.
- **DrawImageUnscaled**: Zeichnet das angegebene Bild in seiner ursprünglichen physischen Größe.
- **DrawImageUnscaledAndClipped**: Zeichnet das angegebene Bild ohne Skalierung und beschneidet es ggf. auf die Größe des angegebenen Rechtecks.
- **DrawLine**: Zeichnet eine Linie.
- **DrawLines**: Zeichnet eine Reihe von Linien.

- **DrawPath**: Zeichnet einen grafischen Pfad:
- **DrawPie**: Zeichnet eine durch eine Ellipse definierte Kreisform.
- **DrawPolygon**: Zeichnet ein Vieleck.
- **DrawRectangle**: Zeichnet ein Rechteck.
- **DrawString**: Zeichnet die angegebene Textzeichenfolge.

Diese Methoden sehen wir uns in den folgenden Abschnitten im Detail an und stellen dabei weitere benötigte Klassen vor wie beispielsweise solche zur Angabe des zu verwendenden Stiftes.

Eine einfache Linie hinzufügen

Den Start machen wir mit einer einfachen Methode, die eine Linie zu einem Bitmap von 480 x 320 Pixeln hinzufügen soll. Im ersten Beispiel nutzen wir die Überladung der **DrawLine**-Methode, welche mit dem ersten Parameter ein **Pen**-Objekt entgegennimmt und mit den übrigen vieren die Koordinaten des Startpunkts und des Endpunkts der zu zeichnenden Linie. Da die Koordinaten **0**-basiert sind, soll unsere diagonale Linie vom Punkt 0,0 zum Punkt 479, 319 verlaufen:

```
Sub EinfacheLinie
    Dim strFilename As String = "EinfacheLinie.bmp"
    Dim bmp As New Bitmap(480, 320)
    Dim objGraphics As Graphics
    Dim objPen As New Pen(color.Black)
    objGraphics = Graphics.FromImage(bmp)
    objGraphics.DrawLine(objPen, 0,0,479,319)
    bmp.Save(strPath & strFilename)
    Process.Start(strPath & strFilename)
End Sub
```

Hier definieren wir außerdem ein **Pen**-Element, bei dessen Initialisierung wir eine Überladung verwenden, welche nur die Farbe des zu verwendenden Stiftes entgegennimmt – hier **Color.Black**. Das mit **objPen** referenzierte **Pen**-Element übergeben wir als ersten Parameter der **DrawLine**-Methode. Anschließend speichern wir das **Bitmap**-Objekt und zeigen es in der entsprechenden Anwendung an – hier in **Paint**. Das Ergebnis sehen sie in Bild 2.

Einfache Linie mit Point-Elementen als Parameter

Gegebenenfalls liegen Ihnen der Start- und der Endpunkt der Linie als Koordinaten bereits vor und Sie möchten diese in Form von **Point**-Objekten erfassen. Dazu erstellen Sie wie folgt zwei neue **Point**-Objekte, und zwar vor dem Aufruf der **DrawLine**-Methode:

```
Dim objStartPoint As New Point(0,0)
Dim objEndpoint As New
Point(479,319)
```

In der **DrawLine**-Methode geben Sie die Punkte aus **objStartpoint** und **objEndpoint** dann als zweiten und dritten Parameter an:

```
objGraphics.DrawLine(objPen, objSt-
artPoint, objEndpoint)
```

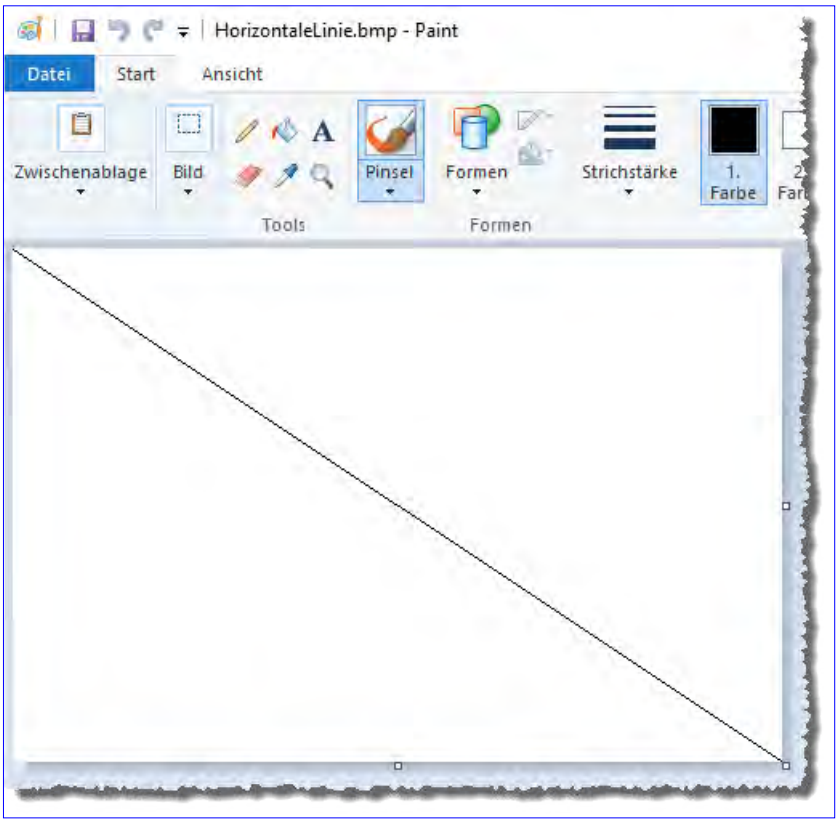


Bild 2: Eine mit der **DrawLine**-Methode erstellte Linie.

Den Stift mit der Pen-Klasse anpassen

Die oben gezeichnete Linie ist eine schwarze, durchgezogene Linie mit einer Breite von einem Pixel. Die Farbe Schwarz haben wir beim Erstellen des **Pen**-Elements als Parameter angegeben, die übrigen Eigenschaften werden standardmäßig eingestellt:

```
Dim objPen As New Pen(Color.Black)
```

Die Breite können Sie mit dem zweiten Parameter einer weiteren Überladung des Konstruktors der **Pen**-Klasse angeben, hier zum Beispiel für eine Linie mit einer Breite von drei Pixeln:

```
Dim objPen As New Pen(Color.Black, 3)
```

Später schauen wir uns weitere Möglichkeiten der **Pen**-Klasse an.

Einen Linienzug zeichnen

Die Methode **DrawLines** erlaubt das Zeichnen mehrerer zusammenhängender Linien gleichzeitig, wobei Sie die Koordinaten der Start- und Endpunkte als **Point**-Elemente jeweils mit einem Array übergeben.

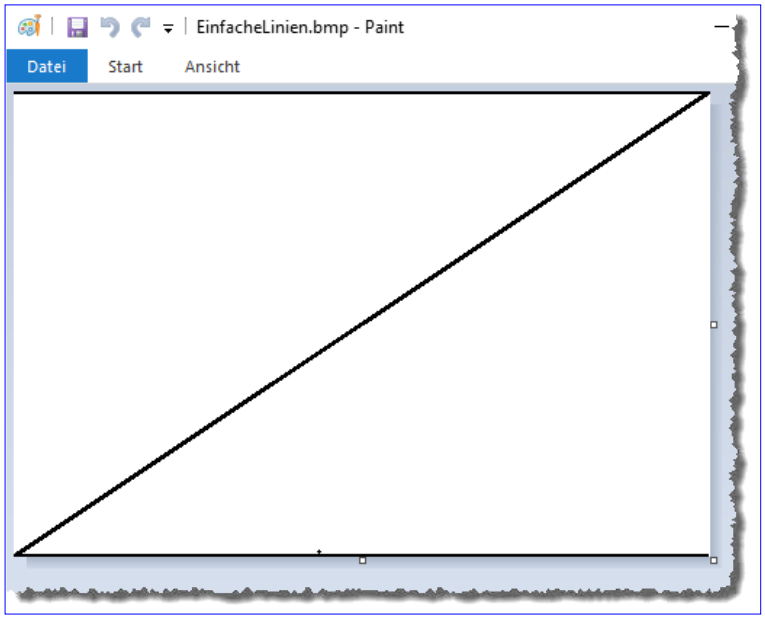


Bild 3: Ein Linienzug in Z-Form

Im folgenden Beispiel erstellen wir wieder ein Bitmap mit 480 x 320 Pixeln und referenzieren dieses mit dem **Graphics**-Element. Dann erstellen wir das Array der Punkte, wobei die Punkte jeweils als neue **Point**-Elemente angeben. Danach rufen wir die **DrawLines**-Methode auf und übergeben dieser als zweiten Parameter das Array aus **objPoints** (siehe Methode **EinfacherLinienzug** in der Beispieldatei):

```
Dim strFilename As String = "EinfacheLinien.bmp"
Dim bmp As New Bitmap(480, 320)
Dim objGraphics As Graphics
Dim objPoints As Point() = {New Point(0,0), New Point(479,0), New Point(0,319), New Point(479,319)}
Dim objPen As New Pen(color.Black, 3)
objGraphics = Graphics.FromImage(bmp)
objGraphics.DrawLines(objPen, objPoints)
bmp.Save(strPath & strFilename)
Process.Start(strPath & strFilename)
```

Das Ergebnis sehen wir in Bild 3. Der resultierende Linienzug liefert ein Z.

Rechtecke zeichnen mit DrawRectangle

Als Nächstes schauen wir uns an, wie wir Rechtecke zeichnen können. Ein Rechteck erfordert prinzipiell genau wie eine Linie nur die Angabe zweier Koordinaten – die des oberen linken und des unteren rechten Punktes. Zum Erstellen eines Rechtecks verwenden wir die Methode **DrawRectangle**.

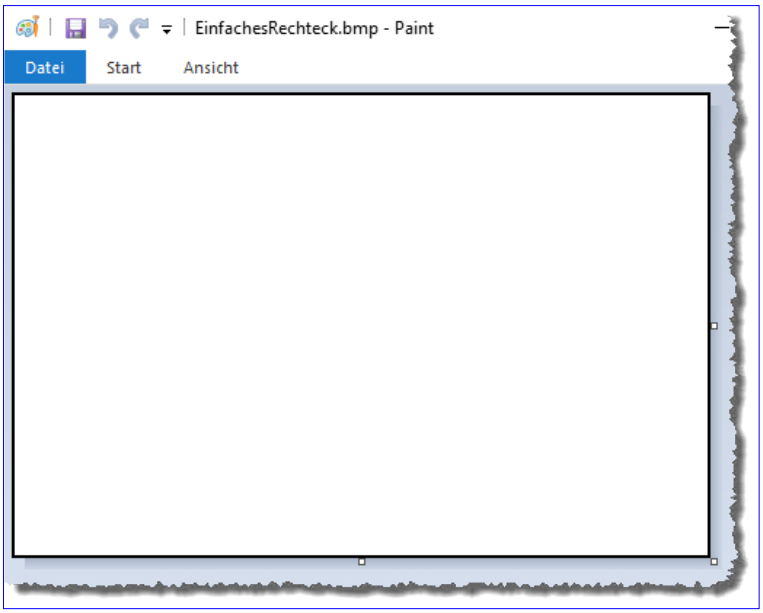


Bild 4: Ein einfaches Rechteck

Diese erwartet in einer Überladung statt der oben angegebenen Koordinaten jedoch nur eine absolute Koordinate für den ersten Punkt mit x und y sowie mit weiteren Parametern die Angabe der gewünschten Breite und Höhe. Das sieht für ein Rechteck um das komplette Bitmap wie folgt aus (siehe Bild 4):

```
Dim strFilename As String = "EinfachesRechteck.bmp"
Dim bmp As New Bitmap(480, 320)
Dim objGraphics As Graphics
Dim objPen As New Pen(color.Black, 3)
objGraphics = Graphics.FromImage(bmp)
objGraphics.DrawRectangle(objPen, 0, 0, 479, 319)
```

Rechteck auf Basis einer Rect-Struktur erstellen

Sie können allerdings auch eine **Rect**-Struktur erstellen, welche prinzipiell die gleichen Informationen enthält, und diese als zweiten Parameter an die **DrawRectangle**-Methode übergeben. Wir haben das Rechteck nun mit einem Abstand von 10 Pixeln zu den Seitenrändern erstellt.

Das Ergebnis sieht wie in Bild 5 aus, die vollständige Methode finden Sie unter dem Namen **EinfachesRechteck_Rect** in der Beispieldatei:

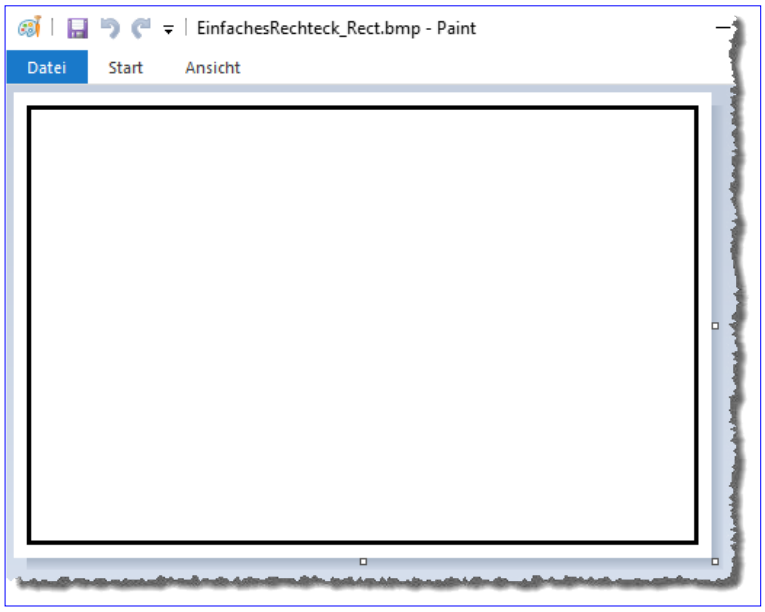


Bild 5: Ein Rechteck mit 10 Pixeln Abstand zum Rand des Bitmaps

```
Dim strFilename As String = "Einfaches-
Rechteck_Rect.bmp"
Dim bmp As New Bitmap(480, 320)
Dim objGraphics As Graphics
Dim objRectangle As New rectangle(10, 10, 459, 299)
Dim objPen As New Pen(color.Black, 3)
objGraphics = Graphics.FromImage(bmp)
objGraphics.DrawRectangle(objPen, objRectangle)
```

Füllen eines Rechtecks mit einer Farbe

Gegebenenfalls möchten Sie das Rechteck auch mit einer Farbe füllen wie in Bild 6.

Bild 6: Gefülltes Rechteck

Wider Erwarten gibt es keine Überladung für die **DrawRectangle**-Methode, welche die Angabe einer Füllfarbe zulässt. Stattdessen gibt es eine weitere Methode namens **FillRectangle**, der wir ebenfalls eine **Rect**-Struktur mit den Abmessungen des Rechtecks übergeben können. Außerdem erwartet diese die Füllfarbe in Form eines **Brush**-Elements. Die **FillRectangle**-Methode füllt allerdings kein zuvor noch zu erstellendes Rechteck, sondern sie erstellt ein neues Rechteck, das direkt mit der angegebenen Farbe gefüllt

wird. Wenn Sie also beispielsweise ein Rechteck mit einem schwarzen Rahmen erstellen wollen, das mit roter Farbe gefüllt ist, benötigen Sie sowohl den Aufruf einer **DrawRectangle**- als auch einer **FillRectangle**-Methode. In diesem Fall definieren wir das Rechteck mit dem schwarzen Rahmen wie zuvor. Damit das rote, ausgefüllte Rechteck genau innerhalb des schwarzen Rechtecks angezeigt wird und den Rahmen nicht überdeckt, beginnt es bei den Koordinaten **x=12** und **y=12** (die Linie des schwarzen Rechtecks startet bei **(10|10)**, aber da wir als Breite 3 Pixel angegeben haben, wird auch noch das Pixel **(11|11)** vom Rand des Rechtecks eingenommen – und natürlich noch die anderen Pixel, die innerhalb der eigentlichen Linie liegen). Die Breite und die Höhe stellen wir ebenfalls entsprechend ein. So erstellen wir also zwei **Rectangle**-Objekte namens **objRectangle** und **objRectangleFill** und nutzen diese dann jeweils als zweiten Parameter der Methoden **DrawRectangle** und **FillRectangle**:

```
Dim strFilename As String = "Rechteck_Gefuellt.bmp"
Dim bmp As New Bitmap(480, 320)
Dim objGraphics As Graphics
Dim objRectangle As New Rectangle(10, 10, 459, 299)
Dim objRectangleFill As New Rectangle(12, 12, 456, 296)
Dim objPen As New Pen(color.Black, 3)
Dim objBrush As New SolidBrush(color.Red)
objGraphics = Graphics.FromImage(bmp)
objGraphics.DrawRectangle(objPen, objRectangle)
objGraphics.FillRectangle(objBrush, objRectangleFill)
```

Zu beachten ist hier noch, dass **FillRectangle** kein **Pen**-, sondern ein **Brush**-Element als ersten Parameter erwartet. **Brush** ist dabei nur eine Schnittstelle, die durch verschiedene Klassen implementiert wird – beispielsweise durch **SolidBrush**, welche wir hier verwenden. Anschließend schauen wir uns verschiedene **Brush**-Ableitungen an.

Füllen mit der Brush-Schnittstelle

Brush ist wie bereits beschrieben nur eine Schnittstelle, Sie können also nicht einfach ein Objekt des Typs **Brush** erstellen wie hier:

```
Dim objBrush As Brush
```

Stattdessen nutzen Sie eine der **Brush**-Ableitungen, von denen die einfachste die **SolidBrush**-Klasse ist. Es gibt beispielsweise noch die folgenden **Brush**-Ableitungen:

- **SolidBrush**: Definiert eine einfache Füllfarbe.
- **LinearGradientBrush**: Definiert zwei Farben und zwei Punkte. Zwischen den beiden angegebenen Punkten gehen die Farben ineinander über.
- **HatchBrush**: Liefert schraffierte Füllmuster.

Füllen mit dem LinearGradientBrush

Um ein Rechteck zu erstellen, das von links oben nach rechts unten mit immer dunkler werdenden Grautönen gefüllt ist, nutzen wir die **Brush**-Implementierung **LinearGradientBrush**. Das Ergebnis soll wie in Bild 7 aussehen.

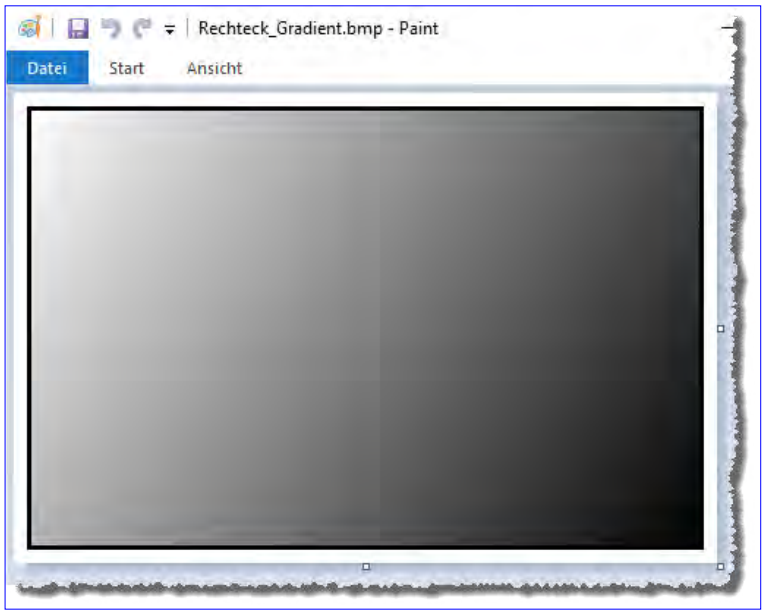


Bild 7: Mit einem Gradienten gefülltes Rechteck

Um dieses zu erreichen, behalten wir die beiden **Rectangle**-Strukturen bei, um wieder einen Rahmen um die auszufüllende Fläche hinzuzufügen. Für das Definieren des **Brush**-Elements auf Basis der Klasse **LinearGradientBrush** benötigen wir zwei **Point**-Elemente, welche angeben, zwischen welchen beiden Punkten der Übergang abgebildet wird.

Wir wählen zwei Punkte, die sich im Rechteck oben links und unten rechts befinden. Diese können sich auch außerhalb des Bereiches befinden, der letztlich durch die **FillRectangle**-Methode gefüllt wird – wenn der Punkt links oben beispielsweise nicht weiß sein soll, sondern beispielsweise hellgrau, können Sie den ersten Punkt auch weiter nach links oben verschieben, etwa nach **(-100|-100)**. Neben den beiden **Point**-Elementen erwartet die **LinearGradientBrush**-Klasse beim Initialisieren die Farbe für den ersten Punkt und die Farbe für den zweiten Punkt. Diese geben wir mit **Color.White** und **Color.Black** an.

Die für die oben abgebildete Füllung benötigten Codezeilen sehen wie folgt aus (siehe Methode **Rechteck_Gradient** in der Beispieldatei):

```
Dim strFilename As String = "Rechteck_Gradient.bmp"
Dim bmp As New Bitmap(480, 320)
Dim objGraphics As Graphics
Dim objRectangle As New Rectangle(10, 10, 459, 299)
Dim objRectangleFill As New Rectangle(12, 12, 456, 296)
Dim objPen As New Pen(color.Black, 3)
Dim objPoint1 As New Point(10,10)
Dim objPoint2 As New Point(469, 309)
Dim objBrush As New LinearGradientBrush(objPoint1, objPoint2, Color.White, Color.Black)
objGraphics = Graphics.FromImage(bmp)
objGraphics.DrawRectangle(objPen, objRectangle)
objGraphics.FillRectangle(objBrush, objRectangleFill)
```

Texte in Bitmaps einfügen

In zwei weiteren Artikeln haben wir beschreiben, wie man mit dem Bitmap- und dem Graphics-Objekt arbeitet, um Bilddateien zu erstellen und diese mit einzelnen Pixeln oder auch mit Linien, Formen und Kurven füllt. Im vorliegenden Artikel gehen wir noch einen Schritt weiter und schauen uns an, wie sich Text in Bitmaps einfügen lassen und welche Möglichkeiten sich für ihre Platzierung und Ausrichtung ergeben.

Vorbereitende Informationen

Die beiden weiteren Artikel heißen **Bitmaps programmieren mit VB.NET** (**www.datenbankentwickler.net/297**) und **Zeichnen in Bitmaps mit der Graphics-Klasse** (**www.datenbankentwickler.net/298**). Hier finden Sie die grundlegenden Informationen und Techniken, auf die wir auch in diesem Artikel zugreifen.

Genau wie in den oben genannten Artikeln verwenden wir auch in diesem Artikel **LINQPad** als Spielwiese zum Ausprobieren der Beispiele – dort finden Sie auch Informationen zur Vorbereitung von LINQPad für das Reproduzieren unserer Beispiele. Wenn Sie LINQPad bereits installiert haben, können Sie einfach die Datei **Graphics-Programmierung_Texte.linq** aus dem Download zu diesem Artikel per Doppelklick öffnen und finden alles Notwendige.

Texte einfügen

Anlass für diesen und die vorherigen Artikel war eine Anfrage eines Kunden, der eigentlich mit Access arbeitet und dort in einem Bericht Label für Behälter ausdrucken sollte, wobei der Text sich in einer Raute befindet. Aus Platzgründen wollte er diese Raute aber um 45° drehen, sodass er ein Quadrat erhält, in dem jedoch der Text um 45° gekippt enthalten ist. Das wiederum kann man mit Access-Berichten nicht abbilden. Die Idee war dann, die komplette Raute mit Text direkt um 45° gekippt in einer Bilddatei abzubilden, die man dann über die Datensatzquelle und ein Bild-Steuerelement in den Bericht einbinden kann.

Welche Möglichkeiten bietet nun also .NET zum Einfügen von Texten in Bilddateien? Schauen wir uns zunächst ein einfaches Beispiel an:

```
Sub TextInBitmap_Simple
    Dim strFilename As String = "TextInBitmap.png"
    Dim bmp As New Bitmap(480, 480)
    Dim objGraphics As Graphics
    objGraphics = Graphics.FromImage(bmp)
    objGraphics.DrawString("Beispieltext", New font("Tahoma", 24), Brushes.Black, 0, 0)
    bmp.Save(strPath & strFilename)
    Process.Start(strPath & strFilename)
End Sub
```

In diesem Beispiel definieren wir als Erstes den Dateinamen für die zu erstellende Datei. Dann erstellen wir ein neues **Bitmap**-Element mit einer Größe von 480x480 Pixeln. Dieses nutzen wir dann als Basis für ein neues **Graphics**-Element, welches das **Bitmap**-Element mit der **FromImage**-Methode referenziert. Anschließend nutzen wir bereits die DrawString-Methode, um einen Text zu dem **Graphics**-Objekt hinzuzufügen. Dabei übergeben wir mit dem ersten Parameter den anzuzeigenden Text, mit dem zweiten ein Objekt, das die Schriftart definiert sowie mit dem dritten den zu verwendenden »Pinsel«. Danach folgt bereits das Speichern des erstellten Bitmap-Objekts als **.png**-Datei. Hier nutzen wir noch die Konstante mit dem Verzeichnis, das wir zum Speichern nutzen und die wir im Modulkopf wie folgt definiert haben:

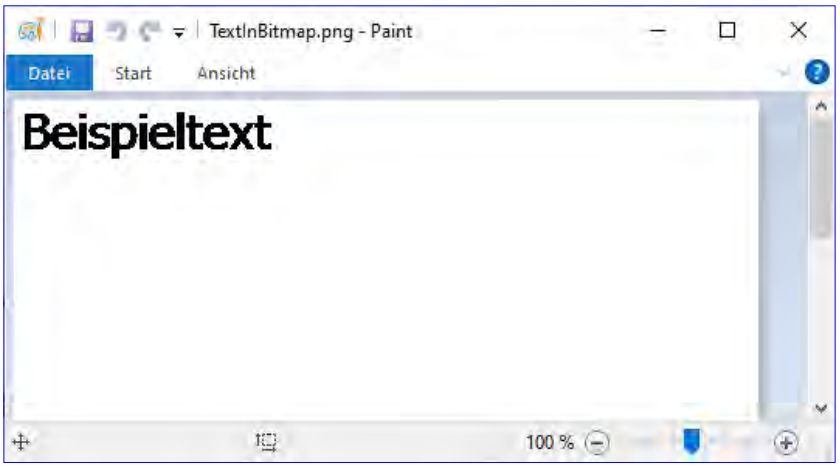


Bild 1: Ausgabe eines ersten Beispiels

```
Const strPath As String = "C:\...\GraphicsTest\"
```

Die letzte Anweisung öffnet die standardmäßig für diesen Dateityp vorgesehene App und zeigt das erstellte Bild darin an (siehe Bild 1).

Die DrawString-Methode

Die **DrawString**-Methode ist das Werkzeug der Klasse **Graphics**, wenn es um die Ausgabe von Texten geht. Sie bietet insgesamt sechs Überladungen an, von denen wir uns die wichtigsten Parameter ansehen:

- **String**: Zeichenfolge, die gezeichnet werden soll
- **Font**: Font-Objekt, das Informationen über die zu verwendende Schriftart, Schriftgröße und weitere Eigenschaften enthält
- **Brush**: Erwartet ein Objekt, das auf einer Implementierung der Brush-Klasse basiert. Im einfachsten Fall nutzt man einfach beispielsweise **Brushes.Black** für einen schwarzen »Pinsel«.
- **x**: x-Koordinate, an welcher der Text platziert werden soll
- **y**: y-Koordinate, an welcher der Text platziert werden soll
- **Format**: Gibt die Formatierung für den zu zeichnenden Text an wie Zeilenabstand, Ausrichtung und so weiter.

Schriftart festlegen

Um die gewünschte Schriftart festzulegen, kann man einfach ein neues Font-Objekt unter Angabe der wichtigsten Parameter erstellen. Für eine fette Schrift des Typs **Courier New** in der Größe 24 sieht das Erstellen eines solchen Objekts wie folgt aus:

```
Dim objFont As New Font("Courier New", 24, FontStyle.Bold)
```

Die Objektvariable **objFont** können wir dann einfach als zweiten Parameter der **DrawString**-Methode nutzen:

```
objGraphics.DrawString("Beispieltext", objFont, Brushes.Black, 0, 0)
```

Brush festlegen

Das **Brush**-Objekt gibt an, in welcher Farbe und mit welchen weiteren »Pinsel«-spezifischen Eigenschaften der Text gezeichnet werden soll. Wenn einfach eine Farbe angegeben wird, mit welcher der Text gezeichnet werden soll, bietet sich eines der Elemente der **Brushes**-Auflistung an. IntelliSense bietet hier alle verfügbaren Farben an, die für den alltäglichen Gebrauch ausreichen sollten (siehe Bild 2).

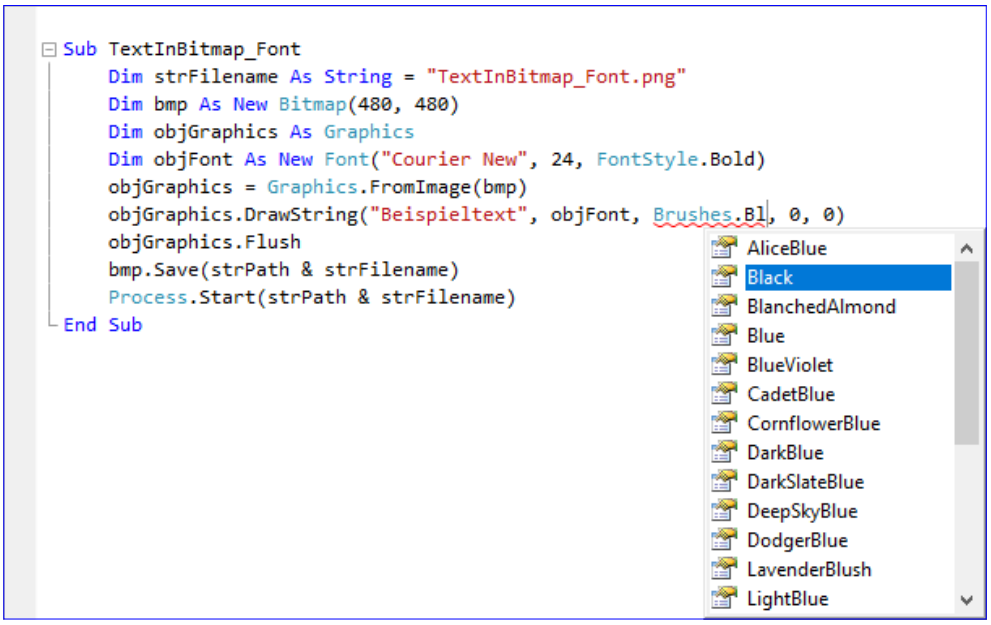


Bild 2: Auswahl eines »Pinsels« über die **Brushes**-Auflistung

Sie können auch eine eigene Objektvariable für das **Brush**-Element deklarieren. Dabei ist wichtig, dass Sie nicht versuchen, die **Brush**-Klasse selbst zu nutzen, da diese nur eine Schnittstellenklasse ist. Die Implementierungen der **Brush**-Klasse wie **HatchBrush**, **LinearGradientBrush** oder **SolidBrush** aber liefern

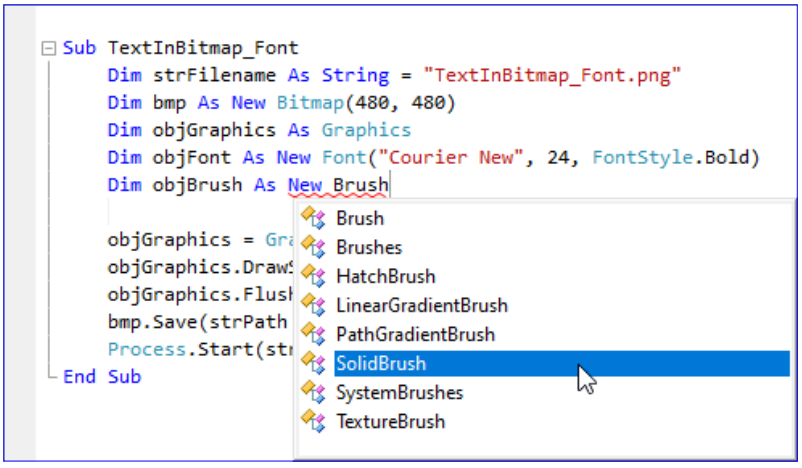


Bild 3: Implementierungen der **Brush**-Klasse

IntelliSense, wenn Sie hinter dem **New**-Schlüsselwort **Brush** eingeben (siehe Bild 3).

Beim Initialisieren einer der **Brush**-Implementierungen mit **New** müssen Sie außerdem die Farbe für den Pinsel angeben. Das können Sie beispielsweise über die Auflistung **Color** erledigen, hier wieder für einen schwarzen Pinsel der Klasse **SolidBrush**:

```
Dim objBrush As New SolidBrush(Color.Black)
```

Bild 4: Schrift mit Farbverlauf

Dieses **SolidBrush**-Objekt weisen Sie dann per Parameter wie folgt der **DrawString**-Methode zu:

```
objGraphics.DrawString("Beispieltext", objFont, objBrush, 0, 0)
```

Alternativ können Sie auch andere Pinsel nutzen, beispielsweise um die Schrift mit einem Farbverlauf anzuzeigen. Wenn die Schrift links schwarz sein soll und dann nach rechts einen Verlauf in Richtung Weiß nehmen soll, nutzen Sie einen Pinsel, der auf Basis der Klasse **LinearGradientBrush** erstellt wird. Der erste und zweite Parameter geben den Start- und Endpunkt des Verlaufs an und der dritte und vierte die Farben für den Verlauf:

```
Dim objBrush As New LinearGradientBrush(New Point(0,0), New Point(480,0), Color.Black, Color.White)
```

Wenn wir die Schriftgröße so anpassen, dass der Text **Beispieltext** sich über den gesamten Farbverlauf mit einer Breite von 480 Pixeln erstreckt, sieht das Ergebnis wie in Bild 4 aus (siehe Methode **TextInBitmap_Font** in der Beispieldatei).

Platzierung des Textes

Die Parameter **x** und **y** geben den linken, oberen Punkt des für den Text benötigten Bereichs an. Hier sind keine weiteren Erläuterungen nötig. Wir müssen uns aber über die Alternativen unterhalten, die in weiteren Überladungen der **DrawString**-Methode zu finden sind. Die erste wirkt sich bezüglich der Positionierung des Textes nicht anders aus als bei der Angabe der Koordinaten von x und y: Sie erwartet statt dieser beiden Parameter eine **Point**-Struktur, welche wiederum die x- und die y-Koordinate enthält.

Eine andere Wirkung, gerade bezogen auf die weiter unten besprochene Ausrichtung des Texts, hat jedoch die Variante mit der Angabe eines **Rect**-Elements. Dieses erwartet die Angabe der x- und y-Koordinate der linken, oberen Ecke sowie die Breite und die Höhe des Bereichs, in dem die Schrift platziert werden soll. Ein Beispiel unter Verwendung eines **Rect**-Elements sieht wie folgt aus:

```
Dim objRect As New Rectangle(20, 20, 440, 440)
...
```

EPC-QR-Code für Rechnungen

Was für eine Erlösung! Nach jahrelanger Quälerei beim Eintippen ewig langer IBANs und Rechnungsnummern in die Banking-App im Smartphone gibt es mittlerweile nicht nur ein Format für einen QR-Code, der alle notwendigen Daten bereitstellt, sondern auch Banken, deren Banking-App das Einlesen von Rechnungsdaten wie den Zahlungsempfänger, IBAN, BIC, Verwendungszweck und Zahlungsbetrag aus einem QR-Code erlauben. Noch schöner: Das Format ist überschaubar und mithilfe einfacher Erweiterungen zum Erstellen von QR-Codes bauen wir uns unsere eigenen EPC-QR-Codes, die wir unseren Rechnungen hinzufügen können. Auf dass unsere Kunden uns feiern, weil sie viel Zeit und Nerven beim Bezahlen unserer Rechnungen sparen!

Der EPC-QR-Code oder GiroCode

Der EPC-QR-Code, der auch als GiroCode bekannt ist, wurde vom European Payments Council standardisiert. Der QR-Code enthält alle notwendigen Informationen für eine SEPA-Überweisung, also beispielsweise für Überweisungen im deutschen Zahlungsraum. Der QR-Code erfasst neben einigen technischen Daten die folgenden Informationen.

Bild 1: Beispiel für einen EPC-QR-Code

Es gibt zwei Versionen, deren Anforderungen sich unwesentlich unterscheiden – hier ist die von uns verwendete mit BIC als Pflichtangabe:

- BIC der Empfängerbank: Verpflichtend nur in Version 1.
- Name des Zahlungsempfängers (maximal 70 Zeichen)
- IBAN des Empfängerkontos
- Zahlungsbetrag im Format **EUR#.##**
- Zweck nach dem DTA-Verfahren (optional)
- Referenz als 35-Zeichen-Code gemäß ISO 11649 RF Creditor Reference (optional)
- Verwendungszweck: maximal 140 Zeichen langer Text (optional)

Ein Beispiel für einen EPC-QR-Code mit den relevanten Informationen sieht wie in Bild 1 aus.

Der in diesem EPC-QR-Code kodierte Text lautet wie folgt:

```
BCD
001
1
SCT
BFSWDE33BER
Spende fuer Wikipedia
DE33100205000001194700
EUR10.00


Wikimedia Foerdergesellschaft
```

In diesem Artikel

In diesem Artikel erstellen wir eine Lösung, mit der wir die Daten, die im EPC-QR-Code abgebildet werden sollen, eingeben und das Ergebnis direkt als Bilddatei im Fenster der Anwendung anzeigen können.

In einem weiteren Artikel namens **EPC-QR-Code per DLL** (**www.datenbankentwickler.net/301**) schauen wir uns an, wie wir eine handliche DLL programmieren können, die wir unter anderem per VBA nutzen können – beispielsweise, um Access-Rechnungsberichte mit diesem EPC-QR-Code zu versehen.

Win-Win

Dadurch, dass der Rechnungsempfänger keine Daten mehr manuell von der Rechnung ablesen und in das Onlineformular einer Überweisung eingeben muss, entstehen zwei wesentliche Vorteile: der Empfänger spart sehr viel Zeit und Empfänger und Rechnungsersteller profitieren davon, dass keine Fehler mehr beim Eingeben der Überweisungsdaten entstehen.

Anlegen des Projekts

Wir erstellen das Projekt als WPF-App mit Visual Basic als Sprache und speichern es unter dem Namen **QRUeberweisung**.

NuGet-Paket zum Erstellen von QR-Codes hinzufügen

Mit einem Klick auf den Menübefehl **Projekt|NuGet-Pakete verwalten...** öffnen wir den **NuGet-Paket-Manager**. Hier suchen wir im Bereich **Durchsuchen** nach Einträgen, die dem Suchbegriff **qr code** entsprechen und finden zum Beispiel den Eintrag **QrCode.Net**, den wir für dieses Projekt nutzen wollen.

Ein Klick auf die Schaltfläche **Installieren** fügt das NuGet-Paket zum Projekt hinzu (siehe Bild 2).

Fenster zur Eingabe der benötigten Daten programmieren

Die Daten wie BIC, IBAN, Empfänger, Verwendungszweck und Betrag wollen wir über ein Fenster eingeben können. Neben den entsprechenden Textfeldern soll das Fenster noch eine Schaltfläche enthalten, mit der wir den QR-Code erstellen.

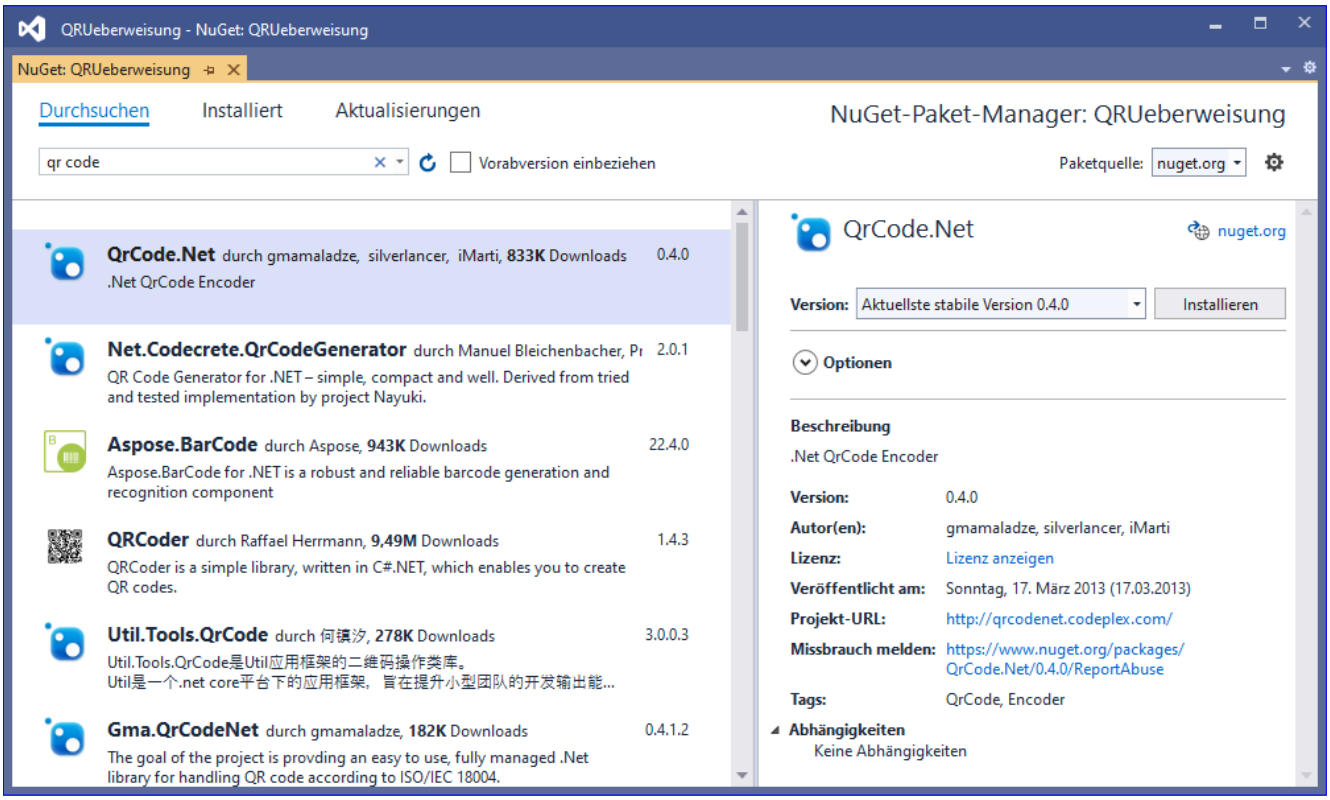


Bild 2: NuGet-Paket zum Erstellen von QR-Codes hinzufügen

Außerdem möchten wir zu Testzwecken schnell ein paar Testdaten in die Textfelder füllen. Das Beispielformular soll schließlich wie in Bild 3 aussehen.

Schließlich wollen wir noch eine Schaltfläche bereitstellen, mit welcher der Benutzer eine Datei mit dem erzeugten EPC-QR-Code auf der Festplatte speichern können soll.

Bild 3: Beispielfenster zum Eingeben und Erstellen von EPC-QR-Codes

Der XAML-Code für das Fenster sieht wie folgt aus, wobei wir hier für eine hübschere Optik ein Icon namens **qr_code.ico** hinzugefügt haben, das im Projekt in einem Ordner namens **images** gespeichert ist:

```
<Window x:Class="MainWindow" ... Title="Beispiel EPC-QR-Code" Height="350" Width="600" WindowStartupLocation="CenterScreen" Icon="Images/qr_code.ico">
```

Damit wir einige Einstellungen nicht für jedes Textfeld und jeden Button definieren müssen, haben wir diese im Bereich **Window.Resources** definiert:

```
<Window.Resources>
    <Style TargetType="TextBox">
        <Setter Property="Margin" Value="5"></Setter>
        <Setter Property="Padding" Value="5"></Setter>
        <Setter Property="HorizontalAlignment" Value="Left"></Setter>
    </Style>
    <Style TargetType="Button">
        <Setter Property="Margin" Value="5"></Setter>
        <Setter Property="Padding" Value="5"></Setter>
        <Setter Property="HorizontalAlignment" Value="Left"></Setter>
    </Style>
</Window.Resources>
```

Die Zeilen und Spalten des Grids haben wir wie folgt definiert, um die Steuerelemente strukturiert anlegen zu können:

```
<Grid>
    <Grid.RowDefinitions>
        <RowDefinition Height="Auto"></RowDefinition>
        <RowDefinition Height="Auto"></RowDefinition>
        <RowDefinition Height="Auto"></RowDefinition>
        <RowDefinition Height="Auto"></RowDefinition>
        <RowDefinition Height="Auto"></RowDefinition>
        <RowDefinition Height="Auto"></RowDefinition>
        <RowDefinition Height="*"></RowDefinition>
    </Grid.RowDefinitions>
    <Grid.ColumnDefinitions>
        <ColumnDefinition Width="Auto"></ColumnDefinition>
        <ColumnDefinition Width="Auto"></ColumnDefinition>
        <ColumnDefinition Width="Auto"></ColumnDefinition>
        <ColumnDefinition Width="*"></ColumnDefinition>
    </Grid.ColumnDefinitions>
```

Steuerelemente des Fensters

Danach folgen bereits die Definitionen für die Steuerelemente.

Das erste ist das Eingabefeld für den Bank Identifier Code (`BIC`) und wird samt Bezeichnungsfeld wie folgt definiert, wobei wir die maximale Anzahl Zeichen auf 11 Zeichen begrenzt haben:

```
<Label>BIC:</Label>
<TextBox x:Name="txtBIC" Grid.Column="1" Width="100" MaxLength="11"></TextBox>
```

Das Textfeld **txtZahlungsempfaenger** für den Zahlungsempfänger darf bis zu 70 Zeichen lang sein:

```
<Label Grid.Row="1">Zahlungsempfänger:</Label>
<TextBox x:Name="txtZahlungsempfaenger" Grid.Row="1" Grid.Column="1" Width="250" MaxLength="70"></
TextBox>
```

Die **International Bank Account Number** (**IBAN**) darf maximal 34 Zeichen enthalten und wird in dem wie folgt definierten Textfeld **txtIBAN** eingegeben:

```
<Label Grid.Row="2">IBAN:</Label>
<TextBox x:Name="txtIBAN" Grid.Row="2" Grid.Column="1" Width="250" MaxLength="34"></TextBox>
```

Für die Eingabe des Betrags haben wir eine Validierung eingebaut, die lediglich die Eingabe von Zahlen und maximal einem Komma erlaubt. Die Validierung erfolgt in der Methode **PreviewTextInput** und die Definition des Textfeldes sieht wie folgt aus, die Methode für das **PreviewTextInput**-Attribut schauen wir uns weiter unten an:

```
<Label Grid.Row="3">Betrag:</Label>
<TextBox x:Name="txtBetrag" Grid.Row="3" Grid.Column="1" Width="100"
    PreviewTextInput="txtBetrag_PreviewTextInput"></TextBox>
```

Auch der Verwendungszweck hat eine begrenzte Zeichenzahl, nämlich 140. Hier können Sie allerdings Zeilenumbrüche einfügen, in diesem Fall per Eingabetaste. Damit die Eingabetaste als Zeilenumbruch erkannt wird, stellen wir für das Textfeld **txtVerwendungszweck** das Attribut **AccepptReturns** auf **True** ein. Auch sollen Zeilen, die über die Textfeldbreite hinausragen, umgebrochen werden, was wir durch den Wert **Wrap** für das Attribut **Wrapping** erreichen:

```
<Label Grid.Row="4">Verwendungszweck:</Label>
<TextBox x:Name="txtVerwendungszweck" Grid.Row="4" Grid.Column="1" Height="100" Width="250" Text-
Wrapping="Wrap" AcceptsReturn="True" MaxLength="140"></TextBox>
```

Schaltflächen des Fensters

Damit kommen wir zu den beiden unteren Schaltflächen, die wir in einem **StackPanel**-Element organisieren, das sich in der fünften Zeile befindet und zwei Spalten umfasst. Die erste Schaltfläche heißt **btnErstellen**, die zweite **btnBeispieldaten**. Auch ihre Ereignismethoden stellen wir weiter unten vor:

EPC-QR-Code per DLL

Das Bezahlen von Rechnungen macht heutzutage noch weniger Spaß als es ohnehin schon tut – neben der unausweichlichen Verminderung des Kontostandes ist es auch keine Freude, ellenlange IBAN-Folgen und kryptische Verwendungszwecke einzugeben. Wie Ihre Kunden demnächst wenigstens letzteres schwungvoll erledigen können, haben wir im Artikel EPC-QR-Code für Rechnungen (www.datenbankentwickler.net/300) gezeigt. Damit gelingt das Erzeugen eines QR-Codes, der alle notwendigen Informationen enthält, die von den gängigen Smartphone-Apps der Banken problemlos per Kamera eingelesen und in die Felder des Überweisungsformulars füllt. Im genannten Artikel können Sie den QR-Code für eine Rechnung vorbereiten, aber wenn Sie mehrere Rechnungen auf einen Rutsch stellen wollen, dann wollen Sie die Daten beispielsweise aus einer Datenbank direkt per Code verarbeiten und den erstellten QR-Code direkt in eine Rechnung einbetten. Die Voraussetzungen erledigen wir im vorliegenden Artikel, indem wir eine DLL mit allen notwendigen Funktionen zum Erstellen des notwendigen QR-Codes programmieren.

Aufgaben der DLL

Was soll diese DLL erledigen? Grob gefasst soll sie die Daten entgegennehmen, die für eine Überweisung benötigt werden, und auf irgendeine Art und Weise den dann zu erstellenden EPC-QR-Code zurückliefern.

Auf welche Weise könnte das geschehen? Dazu gibt es zum Beispiel die folgenden Möglichkeiten:

- Man übergibt beim Initialisieren der DLL einen Pfad zu einer Datei, in welcher der Code dann gespeichert wird.
- Oder die DLL stellt ein direkt weiterverarbeitbares Objekt mit dem entsprechenden Objekttyp zurück – für die Weiterverarbeitung unter Access beispielsweise als **StdPicture**. Hier wäre zu klären, ob .NET dies kann.

Übergabe der benötigten Daten

Welche Werte wollen wir der DLL zum Verarbeiten im EPC-QR-Code übergeben?

- BIC
- IBAN
- Verwendungszweck
- Empfängername
- Betrag

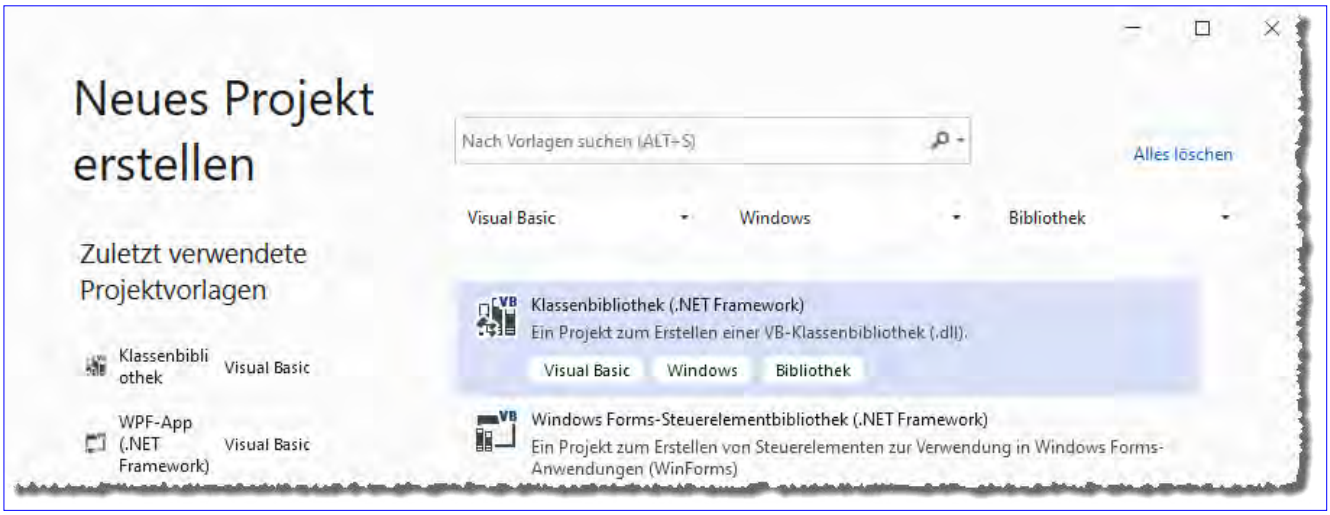


Bild 1: Auswahl des Projekttyps

Art der Übergabe der benötigten Informationen

Wie wollen wir die Informationen an die DLL übergeben? Für diese Informationen können wir entweder Eigenschaften für die in der DLL enthaltenen Klasse, denen wir die Werte zuweisen können, oder wir statten die Funktion, welche den QR-Code erstellt und dann in der einen oder anderen Form zurückliefert, mit den entsprechenden Parametern aus. Wir implementieren einfach beide Varianten, damit der Benutzer die DLL nach seinen eigenen Wünschen einsetzen kann.

Projekt erstellen

Als Erstes erstellen wir ein neues Projekt mit der Vorlage **Klassenbibliothek (.NET Framework)** für die Sprache **Visual Basic** (siehe Bild 1).

Anschließend geben wir als Namen **amvFotoueberweisung** ein und wählen den gewünschten Pfad aus (siehe Bild 2). Hier wählen wir auch das gewünschte Zielframework aus, in diesem Fall **.NET 4.7.2**.

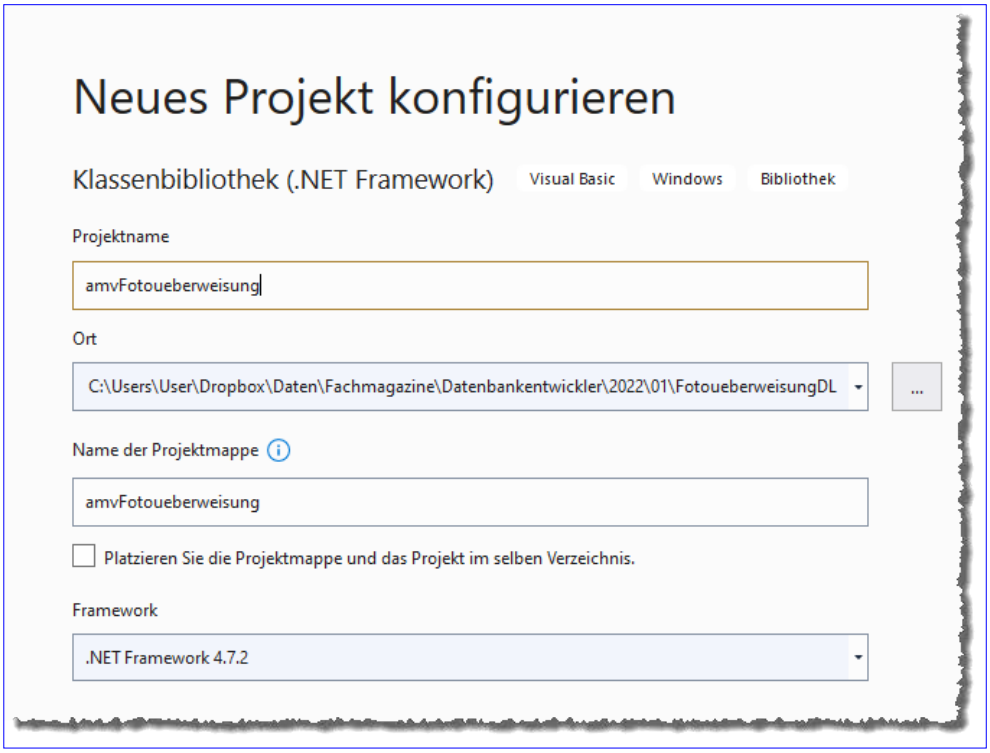


Bild 2: Konfigurieren des neuen Projekts

Klassennamen anpassen

Danach stellen wir den Klassennamen von **Class1.vb** auf **EPCQRCode.vb** um. Die anschließende Nachfrage, ob auch alle Verweise auf die Klasse geändert werden sollen, akzeptieren wir mit einem Klick auf **Ja**.

Öffentliche Eigenschaften hinzufügen

Anschließend erweitern wir die noch leere Klasse um mehrere Elemente, als Erstes um den Import des Namespaces **System.Runtime.InteropServices**:

```
Imports System.Runtime.InteropServices
```

Danach könnten wir bereits die Klasse **EPCQRCode** hinzufügen und für diese die fünf öffentlichen Eigenschaften definieren, die für die Übergabe der für das Erstellen des QR-Codes notwendigen Daten verwendet werden. Wenn wir diese Klasse später im VBA-Editor einer Access-Anwendung referenzieren, zeigt diese Klasse jedoch nicht nur die von uns hinzugefügten Eigenschaften an, sondern auch die Standardeigenschaften einer Klasse wie **GetHashCode**, **GetType** und **ToString**. Diese verwirren in diesem Kontext jedoch mehr als dass sie helfen, also sorgen wir dafür, dass unsere Klasse nur die von uns definierten Member anzeigt.

Dies erreichen wir, indem wir eine öffentliche Schnittstelle namens **IEPCQRCode** definieren, welche nur die gewünschten Eigenschaften vorgibt. Für diese legen wir überdies ein Attribut fest, und zwar **InterfaceType** mit dem Wert **ComInterfaceType.InterfaceIsDual** (dafür benötigen wir den oben genannten **Namespace**-Verweis):

```
<InterfaceType(ComInterfaceType.InterfaceIsDual)>
Public Interface IEPCQRCode
    Property Verwendungszweck As String
    Property BIC As String
    Property IBAN As String
    Property Betrag As <Runtime.InteropServices.MarshalAs( _
        Runtime.InteropServices.UnmanagedType.Currency)> Decimal
    Property Empfaenger As String
End Interface
```

Für den Betrag ist ein kleiner Trick notwendig: Der Datentyp **Decimal** von .NET kann von VBA nicht verarbeitet werden, daher wandeln wir diesen in **Currency** um.

Nun implementieren wir diese Schnittstelle in Form der Klasse **EPCQRCode**, für die wir ebenfalls ein Attribut festlegen, nämlich **ClassInterface** mit dem Wert **ClassInterfaceType.None**. Die Klasse enthält Implementierungen aller in der Schnittstelle festgelegten Eigenschaften, was durch das jeweilige Anhängsel wie **Implements IEPCQRCode.Verwendungszweck** gekennzeichnet wird:

```
<ClassInterface(ClassInterfaceType.None)>
Public Class EPCQRCode
```

```
    Implements IEPCQRCode
    Public Property Verwendungszweck As String Implements IEPCQRCode.Verwendungszweck
    Public Property BIC As String Implements IEPCQRCode.BIC
    Public Property IBAN As String Implements IEPCQRCode.IBAN
    Public Property Betrag As Decimal Implements IEPCQRCode.Betrag
    Public Property Empfaenger As String Implements IEPCQRCode.Empfaenger
End Class
```

Um diese zuerst einmal zu testen, bevor wir die Eigenschaften mit Leben füllen und die Methoden und Funktionen hinzufügen, nehmen wir nun einige weitere Einstellungen am Projekt vor:

- Registrieren für COM-Interop
- Assembly COM-sichtbar machen

Registrieren für COM-Interop

Die Überschrift gibt bereits den Namen der Eigenschaft wieder, die wir aktivieren müssen. Dazu klicken Sie mit der rechten Maustaste auf den Projekteintrag im Projektmappen-Explorer und wählen dort den Eintrag **Eigenschaften** aus.

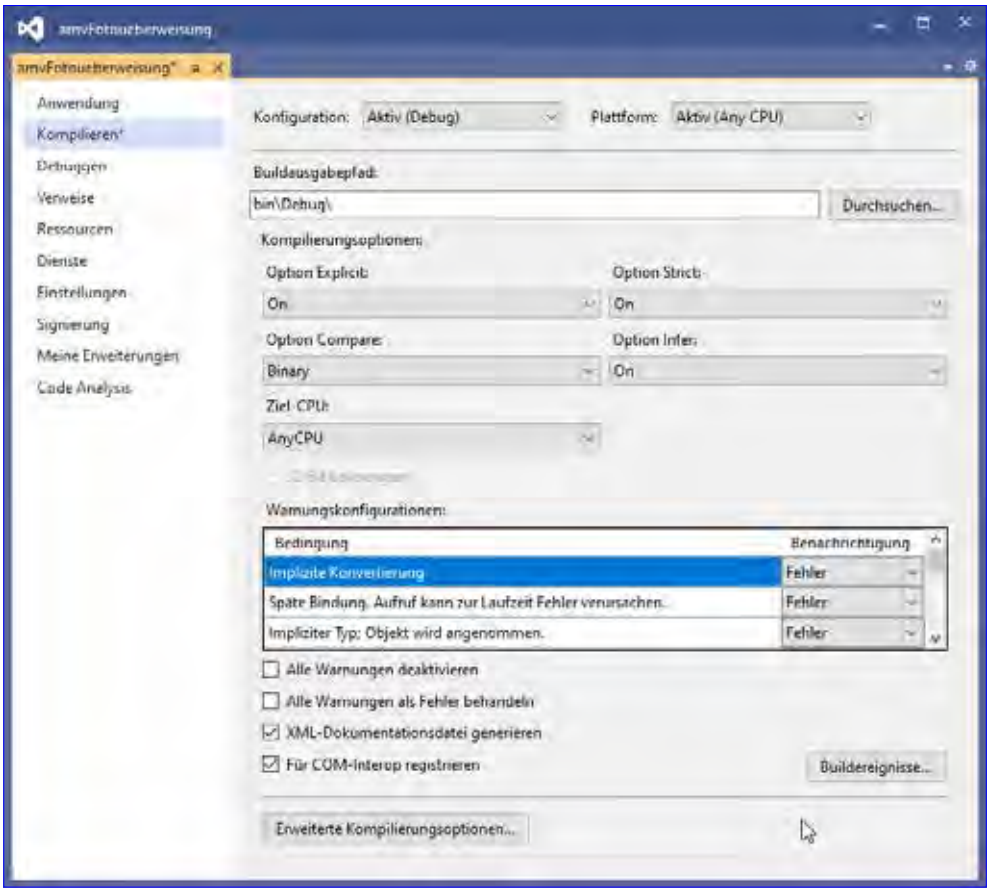


Bild 3: Aktivieren der Eigenschaft **Für COM-Interop registrieren**

Im nun erscheinenden Eigenschaften-Bereich wechseln wir links zum Registerreiter **Kompilieren**. Dort finden wir unten die Eigenschaft **Für COM-Interop registrieren**, die wir nun aktivieren (siehe Bild 3).

Assembly COM-sichtbar machen

Für die zweite wichtige Einstellung wechseln wir links zum Registerreiter **Anwendung**. Hier finden wir die Schaltfläche **As-**

Setup für COM-DLLs mit Inno Setup

Eine COM-DLL mit Visual Studio zu entwickeln ist je nach der Aufgabenstellung schnell erledigt. Ein Klick auf Erstellen und die DLL kann auch schon in die jeweilige Anwendung eingebunden werden. Etwas komplizierter wird es, wenn diese DLL auf einem anderen Rechner installiert werden soll. Dazu reicht es zwar auch aus, die DLL und andere benötigte Dateien auf diesen Rechner zu kopieren und diese mit der App »Regasm.exe« zu registrieren, die auf jedem Rechner vorhanden ist. Allerdings möchte man das dem Kunden nicht unbedingt zumuten. Viel einfacher gelingt dies mit einem Setup, das erstaunlich schnell erstellt ist. Wie Sie ein Setup für die COM-DLL aus dem Artikel EPC-QR-Code per DLL (www.datenbankentwickler.net/301) erstellen, lesen Sie im vorliegenden Artikel.

Wenn Sie eine Lösung auf dem Rechner des Kunden beziehungsweise Benutzers installieren wollen, ist die gute alte **Setup.exe**-Datei immer noch eines der einfachsten Mittel. Wenn man weiß, wie es geht, kann man damit verschiedene Voraussetzungen prüfen, benötigte Verzeichnisse anlegen, Dateien an den gewünschten Ort kopieren und Einträge in die Registry vornehmen, soweit notwendig. Das sind auch gleich die Aufgaben, die wir in diesem Artikel besprechen werden.

Wir gehen an dieser Stelle nicht ausführlich auf Inno Setup ein, sondern zeigen nur, wie Sie das Setup damit erstellen können. Wir nutzen dabei die kostenlose Version von Inno Setup, die Sie unter folgendem Link finden:

```
https://jrsoftware.org/
```

Nachdem Sie Inno Setup installiert haben, können Sie es gleich starten. Der Inno Setup Compiler präsentiert sich dann mit einem Fenster wie in Bild 1. Hier haben wir allerdings bereits den Code für unser Setup eingefügt.

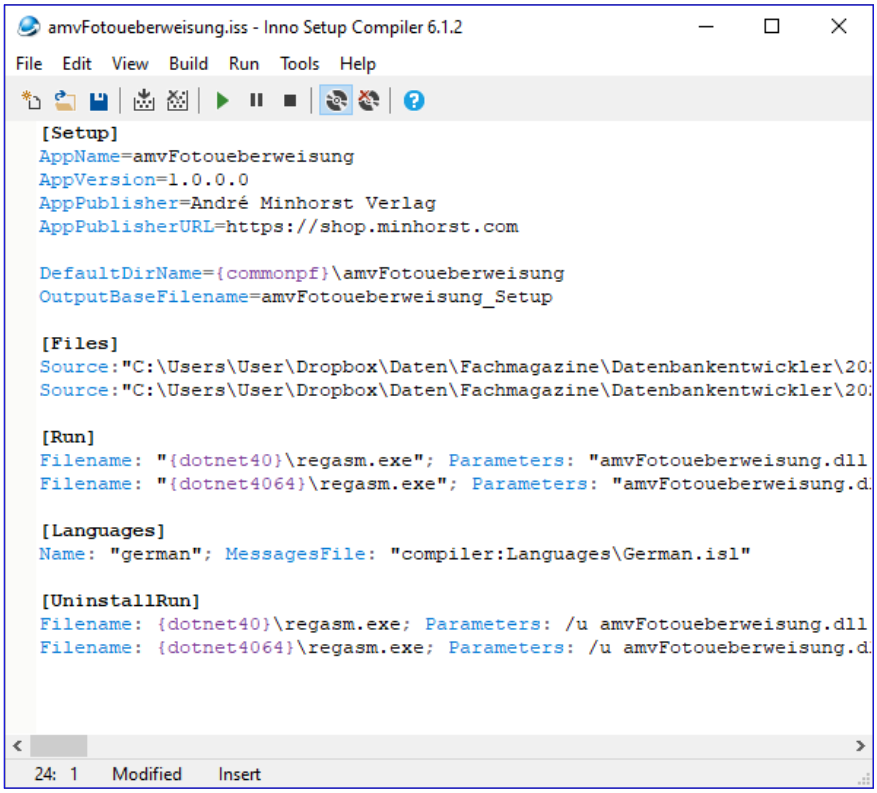


Bild 1: Der Inno Setup Compiler

Code des Setups

Diesen Code schauen wir uns als Nächstes an. Das Setup besteht aus mehreren Bereichen, in diesem Fall

Setup, **Files**, **Run**, **Languages** und **UninstallRun**. Der erste Teil namens **[Setup]** enthält die wichtigsten Informationen, in diesem Fall Name und Version der Anwendung, Name und URL des Herstellers sowie das standardmäßig zu verwendende Verzeichnis für die Installation und den Namen der zu erstellenden Setup-Datei ohne Dateiendung:

```
[Setup]
AppName=amvFotoueberweisung
AppVersion=1.0.0.0
AppPublisher=André Minhorst Verlag
AppPublisherURL=https://shop.minhorst.com

DefaultDirName={commonpf}\amvFotoueberweisung
OutputBaseFilename=amvFotoueberweisung_Setup
```

Danach folgt der Teil **[Files]** mit den Dateien, die dem Setup hinzugefügt werden sollen. Hier geben wir sowohl den Pfad an, unter dem wir die Dateien finden sowie das Zielverzeichnis. Dieses wird hier mit dem Platzhalter **{app}** angegeben, was dem **Programme**-Verzeichnis unter Windows entspricht:

```
[Files]
Source:"C:\...\amvFotoueberweisung.dll"; DestDir: "{app}"
Source:"C:\...\Gma.QrCodeNet.Encoding.dll"; DestDir: "{app}"
```

Der **[Run]**-Teil enthält Aufrufe von Tools, die bei der Installation ausgeführt werden sollen. In diesem Fall rufen wir das Programm **regasm.exe** auf, welches das Erstellen einer **.tlb**-Datei und das Registrieren der DLL durchführt. In Zeiten von 32-Bit und 64-Bit-Windows fragen wir noch ab, welche Variante vorliegt und rufen die entsprechende Version der App **regasm.exe** auf:

```
[Run]
```